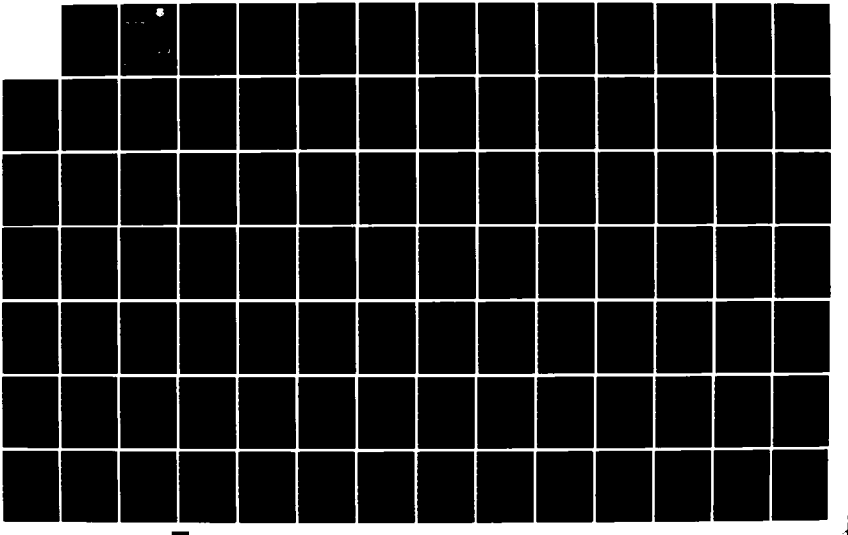
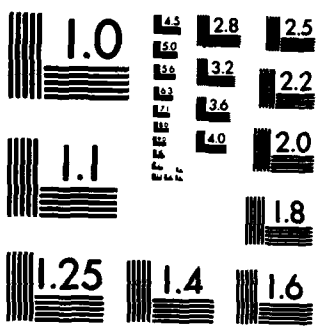
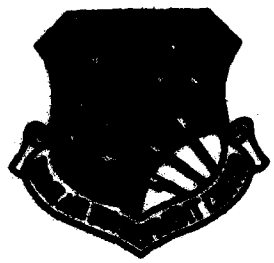
AD-A146 737 NSW (NATIONAL SOFTWARE WORKS) LESSONS LEARNED(U) 1/2
MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD
C MUNTZ ET AL. MAY 84 CADD-8312-3001 RADC-TR-84-90
UNCLASSIFIED F30602-83-C-0040 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0

RADC-TR-84-90
Final Technical Report
May 1984



(12)

NSW LESSONS LEARNED

Massachusetts Computer Associates, Inc.

Charles Muntz, Mark Marcus, Kirk Sattley and Bonnie Shipman

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-84-90 has been reviewed and is approved for publication.

APPROVED: Patricia J. Baskinger

PATRICIA J. BASKINGER
Project Engineer

APPROVED: R Raposo

RONALD S. RAPOSO
Acting Chief, Command & Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A146736

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CADD-8312-3001	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-84-90
6a. NAME OF PERFORMING ORGANIZATION Massachusetts Computer Associates, Inc. — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State and ZIP Code) 26 Princess Street Wakefield MA 01880	7b. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center — 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-83-C-0040
8c. ADDRESS (City, State and ZIP Code) Griffiss AFB NY 13441	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
63728F	2531	01	11
64740F	2526	07	03

11. TITLE (Include Security Classification)
NSW LESSONS LEARNED

12. PERSONAL AUTHOR(S)
Charles Muntz, Mark Marcus, Kirk Sattley, Bonnie Shipman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM Oct 82 TO Dec 83	May 1984	114

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	National Software Works, Distributed Systems, NSW, ARPA Network, Network Operating Systems
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The goals of the National Software Works Project were to demonstrate the feasibility and practicality of providing users "uniform access" to services on a diverse set of computers and operating systems. We have, indeed, succeeded in showing that such access is possible by building, running, and maintaining a system called NSW, which ran on the Arpanet in one form or another between 1976 and 1983. In its final version, NSW connected an IBM 370, two TOPS-20s, a MULTICS and several UNIXes.

The need to interact with different computer systems is ever increasing in both military and commercial sectors. The lessons learned from the construction of NSW can help those building the next generation of these systems. This report is about those lessons we learned.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

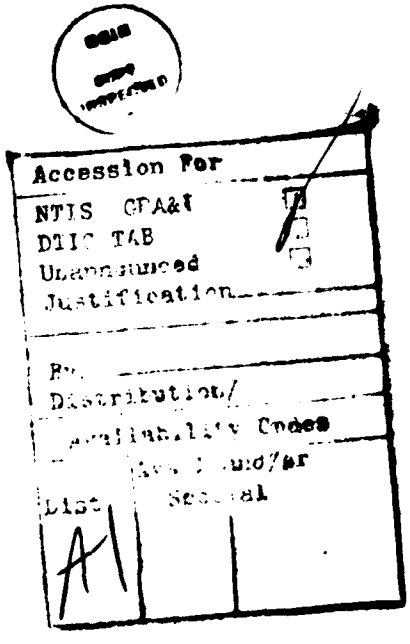
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Patricia J. Baskinger	(315) 330-2805	RADC (COTD)

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

Table of Contents



Preface

The National Software Works Project has come to an end. Its goals were to demonstrate the feasibility and practicality of providing users "uniform access" to services on a diverse set of computers and operating systems. We have, indeed, succeeded in showing that such access is possible by building, running, and maintaining a system called NSW, which ran on the Arpanet in one form or another between 1976 and 1983. In its final version, NSW connected an IBM 370, two TOPS-20s, a Multics, and several UNIXes.

We still believe that the need to interact with different computer systems is ever increasing in both military and commercial sectors; and also that the lessons learned from the construction of NSW can help those building the next generation of these systems. This report is about those lessons we learned.

The structure of this report is:

1. A survey of general systems in industry, government, and academia that facilitate the sharing of computer resources residing on physically distinct processors connected by a network.

2. Considerations in the utility, operation, and management of a heterogenous distributed operating system.

3. Advice to the would-be designer and builder of a distributed heterogeneous operation system.

4. And lastly, a discussion of the NSW technique employed for organizing its (potentially) enormous name space.

Many people from many different organizations have led to the successful design and implementation of the National Software Works Project. From Masachusetts Computer Associates were David Bearisto, Regina Bolduc, Paul Cashman, Janie Cotreau, Ross Faneuf, Dennis Geller, Michael Guerrieri, Kathy Knobe, Leslie Lamport, Mark Marcus, Collen Marcotte, Robert Millstein, Charles

Muntz, David Presberg, Kate Russell, Kirk Sattley, Stuart Schaffner, Bonnie Shipman, Suzanne Sluizer, Nina Tsao, Steve Warshall, and Michael Wolfberg.

From BBN were Jim Calvin, Paul Johnson, William McGregor, Richard Schantz, Steven Swernofsky, Robert Thomas, Stephen Toner, and Frank Ulmer.

From Meta Information Applications, Inc. was Robert Shapiro.

From UCLA were Bob Braden, Neil Ludlam, and Denis De La Roca.

From Honeywell were John Ata and Richard Sheton.

From IITRI were Cliff Carroll, John Dobmeier, and Carol Proctor.

From MIT was Doug Wells.

From SRI were Charles Irby, John Postel, and James White.

From GSG were Frank Bamburger, Brian Bauer, Ross Gale, Pete Kneiss, Doug Payne, and Norm Rasmussen.

For their very fine support, we would like to thank Richard Robinson, Patricia Baskinger and Richard Metzger of Rome Air Development Center.

We would also like to thank the many Air Force Logistic Command users whose many comments have helped shape NSW and this report.

And finally, we would like to thank Paul Cashman, Lyn Churchill, David Presberg, and Steve Schuman for their helpful reviews of this paper.

Massachusetts Computer Associates
December 1983

Preface to the Second Edition

In this edition the page-numbering scheme has changed, and also changes in wording have been made here and there.

Massachusetts Computer Associates
April 1984

1. Survey of Distributed Systems -- by Bonnie Shipman

1.1 ABOUT THE SURVEY

The purpose of this survey is to explore general systems in industry, government, and academia that facilitate the sharing of computer resources residing on physically distinct processors connected by a network. I am explicitly not interested in addressing the subject of multi-computers, where more than one processor communicates through shared memory.

The systems described here present various approaches to the resource-sharing solution. This presentation should also familiarize the reader with computer network terminology, and with various motivations for networking. It provides a taxonomy of network systems, addresses important issues in network operating system (NOS) design, and investigates ideas for future directions.

1.2 RESOURCE SHARING IN CENTRALIZED AND DISTRIBUTED SYSTEMS

The need to share computer hardware and software resources has long been recognized. This need is motivated by at least four factors [Lister 79]:

1. Sharing Cost:

   It is costly and therefore impractical to provide all necessary resources to each user separately.

2. Building on the Work of Others:

   Time and effort are saved by using other people's programs or routines.

3. Sharing Data:

   The same database may be used for several different programs.

4. Removing Redundancy:

It is economical to share a single copy of a program (e.g., a compiler) between several users, rather than to provide a separate copy for each user.

In fact one of the main purposes of an operating system is to facilitate sharing by providing resource allocation, concurrent, not simultaneous access to data, program execution, and protection against corruption. These functions are clearly provided in single computers by operating systems that support multiprogramming.

The potential for resource sharing has increased dramatically with the advent of computer interconnection technology. Distributing systems over computer networks can result in the following advantages [Lantz 80]:

1. The repertoire of available resources can be increased by making new host types accessible via the network.

2. Performance can be enhanced through parallelism, load balancing and functional specialization, and through the placement of processors near sensing devices and near devices to be controlled.

3. Reliability and flexibility can be attained through redundancy in communication paths and processing elements, and through modularity of design.

4. Decomposing a problem solution into subtasks enables handling of increassed complexity of the problem.

5. Costs can be reduced in two areas. Communication costs can be cut by preprocessing data for transmission (thus lowering communication bandwidth requirements) and by placing computing elements near the data to reduce the distance data must be transmitted. Processing costs can be lowered by using cheaper, less complex processing elements which can be mass produced, and through load sharing (allowing relatively idle processors to handle some of the work of a busy processing element).

1.3 A TAXONOMY OF NETWORKS AND DISTRIBUTED SYSTEMS

Computer networks come in several forms (ala [Lantz 80]). Remote-communication networks simply move information from one place to another. Resource-sharing networks allow resources on one computer system to be shared by other systems in order to reduce costs and to provide remote access. Distributed-processing networks allow several autonomous computer systems to solve problems by the division of labor or by functional specialization. As such, distributed-processing networks are natural extensions of real-time multiprogramming systems. Homogeneous networks tie together host computers that share a common architecture; heterogeneous networks are composed of different types of hosts. The component hosts may be geographically distant (longhaul) or close (shorthaul or Local Area Network). Several networks can be connected together through gateway machines to form an internetwork.

Distributed systems are referred to as being tightly coupled if the processors share memory or have centralized resource management. Two examples of this type of system are the CRAY-1 and the Illiac-IV. In loosely coupled distributed systems, processors share no memory and are completely autonomous; the Arpanet is a good example of this. Found in between are multi-computers such as Hydra and Cm*.

1.4 APPROACHES TO RESOURCE SHARING IN DISTRIBUTED SYSTEMS

A very common network architecture consists of one or more function-oriented protocols built on top of a network-wide interprocess communication (IPC) facility, and implemented at every host in the network. Each such protocol specifies rules for dialogue between a "server process" providing a service, and a "user process" requesting that service. Examples of function-oriented protocols are the Arpanet's Telecommunications Network protocol (TELNET) and its File Transfer Protocol (FTP).

TELNET essentially enables a terminal on one computer (computer A) to be attached to another computer (computer B) such that terminal A on computer A functions as if it were local to computer B. This type of protocol is often referred to as "virtual terminal" protocol. FTP specifies conventions for the movement of files between hosts on the network. Service commands supported allow a user to

- GET a remote file to a local file,

- SEND a local file to a remote file,

- APPEND a local file to a remote file,

- RENAME a file on a remote host,

- DELETE a remote file, and

- get a DIRECTORY listing from the remote host.

The protocol specifies the necessary manipulations required to convert a file from the internal representation of the sending machine to a number of intermediate representations which may be easily converted to the internal representation of the receiving machine.

The type of resource sharing made possible by virtual terminal and file transfer protocols is somewhat limited. Virtual terminal protocol enables a terminal user to use several computers, but he must have a login account at each and be familiar with the command language, naming and access conventions of every host he wants to use [White 76] [Millstein 77] [Watson 79] [Watson 83]. A user of FTP may have to select different conversion options depending on a file's intended uses on the receiving host. Furthermore, there is no abstraction of common services required to do networking, so that every distributed application must implement these functions itself.

James E. White [White 76] recognized that effective resource sharing depends upon remote resources being accessible to programs as well as to human users. He proposed that a high-level application-independent framework be devised to aid in the construction of distributed systems in resource-sharing networks. This framework consists of a network-wide protocol for invoking arbitrary named functions on remote processors, and machine-dependent systems software that interfaces one program to another via the protocol. In addition, White suggests that remote functions be thought of as remotely callable subroutines or procedures. This view has two very appealing consequences: it gracefully extends the local programming environment to include remote resources, and it presents a comfortable model to

programmers. Both effects encourage the development of distributed systems, with a resulting increase in sharing. White cautions, however, that application programmers should be aware that...

- Local procedure calls are cheap; remote procedure calls are not.
- Conventional programs usually have a single locus of control; distributed programs need not.
- By no means are all useful forms of network communication modelled by the remote procedure call. The lower level IPC facility must be used in those applications where the procedure call model is inappropriate, i.e., in cases where communication is not transaction-oriented.

White's ideas provide the principles underlying the Xerox Network Systems (XNS) Courier Remote Procedure Call Protocol.

Application programmers benefit from even higher-level network protocols or services. One such service is provided by a battery of library or system routines that automate network access and control functions. An example of this is the ADAPT/DEC ("Advanced Distributed Application Programming Tools") collection of software developed at Digital Equipment Corporation [Peebles 80]. ADAPT is intended to simplify the construction of distributed systems and runs on top of DEC's VAX/VMS computer system.

The use of multiple heterogeneous hosts is made easier if they can be accessed uniformly. In its simplest form a Network Access Machine (NAM) acts as a command processor. It translates user commands like: AT HOST X DO COMMAND Y to a text file in the command language of the appropriate host, and then sends it there for execution [Lantz 80] [Tanenbaum 81]. The result is that the end-user is spared from having to know the idiosyncrasies of every system he wishes to use. A NAM offers these services without requiring modifications to the hardware or operating systems of the participating host computers, but can interact with these hosts at the user interface only. A more general solution to resource-sharing is the Network Operating System.

A Network Operating System (NOS) allows both users and programs to access resources without regard to their physical location [Lantz 80]. A network operating system plays a dual role [Watson 79]. It solves the same problems for a set of computers that a traditional operating system solves for a single processor. An NOS turns a collection of hardware and software resources into a coherent set of abstract objects, and supports their naming, access, sharing, protection, synchronization and intercommunication, including error recovery. It multiplexes and allocates these resources among many concurrent applications. In addition, a network operating system must deal with the problems that arise from the distributed nature of the system, and possibly from the heterogeneity of its component systems. These include translation problems, support of distributed service and resource structures, potentially more difficult error recovery, multiple copy file or database update problems, and multiple controlling administrations. Special efficiency problems must be addressed and solved.

There are a number of important issues that can be used to characterize distributed systems. These are described below (after [Lantz 80]).

1. General Purpose Vs. Special Purpose

   Is the system oriented toward general computer resource sharing, or is it intended to provide limited sharing for a particular application?

2. Guest Level Vs. Base Level Implementation

   Is the NOS layered on top of existing computer operating systems, or built right on the bare hardware? In his book, Computer Networks [Tanenbaum 81], Andrew Tanenbaum distinguishes two categories of network-wide operating systems. In what he calls a network operating system, each host continues to run its old (non-network) system, with the network operating system implemented as a collection of user programs running on various hosts. Examples of this type of system are The National Software Works and the Desperanto System. (Both are described later.) A major advantage of this approach is that it does not invalidate existing software. The other approach is

to throw away existing operating systems on constituent hosts and start from scratch with a single homogeneous distributed operating system. In a distributed operating system all activity is NOS activity; a guest-level network operating system allows both NOS and non-NOS activities.

3. Network Services Provided

It is helpful to recognize that network services can be provided to users on two fronts. Services available to end-users at system monitor level can be distinguished from services provided to application programs as system calls or library routines. Some systems we will look at support one type of interface and not the other, but most provide both.

4. Visibility of Distribution

How obvious is the distributed nature of the system to the user? Does the system allow the user to access local and remote resources transparently? Transparent access makes programming easier, and means programs needn't be changed when resources move around, but may also seduce developers of distributed systems into making expensive remote calls needlessly. What naming scheme is used to identify objects or resources in the system? Global location-independent object names conceal the network, and require some sort of name server to map them into real addresses. If the name server is centralized it may introduce overhead with each file open, and can become a performance bottleneck. In addition, the system is subject to instability because the name server could be vulnerable to failure. Replication and/or distribution of the name server can ease reliability and performance problems, at the cost of implementation complexity [Rowe 82]. A common alternative to host-independent object names is a hierarchical naming scheme where host identifiers are included in all object names. This strategy makes no attempt to hide the system's distributed property. To what degree are failures in components of the distributed system visible to the user? Do error messages to users have to be based upon distributed systems notions (e.g., "The terminal handler is up, but the editor is down")?

5. Reliability

Reliability measures are generally expensive, but necessary for any production system. Replication of control functions such as the name servers, as well as of critical databases increases reliability and availability, but at the cost of complex consistency procedures.

6. Protection

What type of protection is provided for system objects? Is access control associated with an object or a user? How is protection managed on a global basis, e.g., how are capabilities protected?

7. Resource Selection

Who makes resource selection -- the user, the system, or both in combination? In the latter cases, is selection based on dynamic network conditions or static user profiles? Perhaps a system administrator has the authority to change the definition of a command to use a resource on host X instead of on host Y, in a fashion analogous to database administration practice.

8. Resource Management

When resource management is logically and physically centralized, the effect of the distributed environment is minimized and implementation is simplified. But the system then becomes vulnerable to failure of the controlling site, and performance can degrade because of required interhost communication every time the control function is required. Both of these problems can be reduced if the control function is replicated at many sites. Alternatively, the control function can be fully distributed by supplying every entity with its own implementation of the function. This breaks the dependence of one host on another and provides better performance, but adds complexity to the system design, implementation and operation.

9. Extensibility

A network operating system should not be expected to spring full-blown into existence [Watson 79] [Fletcher 80], but instead be designed to encourage new application development. Each new application should not have to tackle anew the problems associated with distributed implementation; the system should provide an implementation layer of functions common to a broad range of applications. Also under this heading comes the question of host participation. Can hosts participate in the NOS in varying degrees?

10. System Model

There are two distinct models associated with distributed operating systems [Tanenbaum 81]. In the object model, the world consists of various objects, each of which has a type, a representation, and a set of operations that can be invoked on it. To be able to invoke an operation on an object, a user process must possess a capability (permission) for that object. It is the duty of the operating system to manage capabilities and to allow operations to be carried out on objects. In the process model, each resource is managed by some server process, and all the operating system does is manage interprocess communication. In many cases servers can run as normal user processes. There are two distinct interprocess communication (IPC) schemes: message passing and function or procedure calls. When message passing IPC primitives are provided, any process can communicate with any other process by exchanging messages. The procedure call model of IPC can be built upon message passing, and provides the user with the same semantics used in local procedure calls. In both cases synchronization is provided by blocking sends and procedure calls (sender or caller suspends execution until the receiver or called procedure finishes and returns control), and parallel execution can be achieved by nonblocking sends or procedure calls. In the latter case either an interrupt mechanism is needed to alert the sender (caller) that the operation has finished, or the sender (caller) must check a status indicator for procedure completion.

11. Administration

Is administration centralized or distributed among the participating hosts? Must a user have a private account on each host?

12. Error Recovery

How does the system respond to hardware and software failures? Does it support atomic transactions (i.e., transactions that are guaranteed to be performed to completion or not at all)?

1.5 SELECTED SYSTEMS

The task of categorizing systems for the network connection of processors on shared computer resources is rendered difficult by the fact that features of such systems overlap. Such systems still seem to fall into different categories which we have designated for the purposes of discussion as follows.

- Extensions of centralized operating systems,
- Systems involving sharing among heterogeneous hosts,
- Systems with decentralized resource management,
- Communication-oriented systems, and
- Integrated systems.

1.5.1 Generalizations of Centralized Operating Systems

A network operating system is sometimes devised as an extension or generalization of a popular centralized operating system. This approach minimizes the amount of support software that must be developed, and at the same time eases transition from a single-host to a network environment [Rowe 82]. Examples of this approach given here include the RSEXEC and COCANET UNIX.

1.5.1.1 RSEXEC

RSEXEC [Thomas 73], developed by Bolt, Beranek and Newman, Inc. during the early 1970's, is a Resource Sharing Executive for

homogeneous TENEX hosts on the Arpanet. (TENEX is the basis for today's TOPS-20 operating system.) It is implemented thereon as a family of guest-level processes with no priviledged code. RSEXEC increases the pool of computer resources available to users to include those of all TENEX hosts on the network. The system includes a command language interpreter (syntactically similar to the single-host TENEX EXEC) which extends the effect of user commands to all such TENEX hosts, and a monitor call interpreter that likewise extends the effect of system calls. Together these end-user and program interfaces give the user uniform access to local and remote resources, and isolate him from having to deal with network protocols directly.

The RSEXEC file system implements a file name space that spans host boundaries, enabling users to reference files on any participating TENEX host. File names are hierarchical pathnames, in the TENEX style, extended to include host names. RSEXEC provides a mechanism called a "user profile" wherein users can specify context information to be used by the system for partial pathname recognition.

The user profile can also include a list of file directories that, taken together, define a user's "composite directory". Directory information for each file listed in each of the component directories is periodically copied to the local site, after acquiring it at the start of a user session. These files can be accessed without incurring the overhead of a remote directory search. When files outside the range of the composite directory are referenced, remote directory information is acquired by RSEXEC.

Users can take advantage of the distributed nature of the system by having RSEXEC maintain images of files at multiple hosts. In fact, RSEXEC was used as a tool for distributing software updates to the family of Arpanet TENEX systems.

1.5.1.2 COCANET UNIX

The COCANET is a local area network at the University of California, Berkeley, that is designed to provide shared access to computer resources available on several different computers in the Department of Electrical Engineering and Computer Science.

Hosts connected to the network are DEC PDP-11s and VAXs running a variation of the UNIX[1] operating system [Ritchie 78] called COCANET UNIX [Rowe 82].

Standard UNIX has been altered in such a way that a user logged in at a host in the network can access all network resources using the same commands and programs that were used in a single processor environment. COCANET UNIX extends the UNIX file system name space by including the host name in each resource name, so a user can refer to a remote file by giving its local hierarchical pathname preceded by the name of the host on which it resides. He can change his working directory to one that is remote with the same command he used to change between local directories in single-site UNIX. Existing programs -- those developed under centralized UNIX -- can run on COCANET UNIX without coding changes. Local and remote files alike can be accessed with the standard UNIX file abstraction. Process creation primitives (i.e., 'fork' followed by an 'exec') have been generalized to permit the execution of a remote program, and the standard UNIX interprocess communication mechanism (pipes) has been modified to permit communication with remote processes.

In addition to supporting existing UNIX applications, COCANET UNIX incorporates a small number of new features that facilitate the development of distributed applications. Several specialized message-oriented IPC mechanisms that support communication within a single processor or among distributed processors, have been introduced to make up for the shortcomings of standard UNIX pipes in a distributed environment, namely:

- Low throughput,
- Only processes with a common parent can communicate, and
- Processes block on reads when no data is available.

---

[1] UNIX is a trademark of Western Electric

Three forms of interprocess communication are supported. A process-to-process "datagram" service supports transaction-oriented communication, useful when one or a few messages are sent to a well-known server process requesting a service on behalf of a user. A process-to-process virtual circuit service ("unicast") supports longer conversations between processes, and is similar to a UNIX pipe. A "multicast" channel transmits a single message to multiple destinations.

COCANET UNIX provides its component hosts with "site autonomy", meaning that each host controls access to its own resources. Security comparable to that of standard UNIX is provided by requiring an explicit login procedure to a remote machine the first time its resources are requested. A simple recovery mechanism is used to detect when a remote machine crashes and to notify all processes which were using resources on the machine that it has crashed.

1.5.2 Systems with Decentralized Resource Management

It is the philosophy of some (including [Clark 80]) that adequate sharing can be provided without global resource naming and management, and without support for elaborate features like replicated files. Two examples of systems that embrace this philosophy are the Apollo Domain System and Rochester's Intelligent Gateway.

1.5.2.1 Apollo Domain

The Apollo Domain System [Apollo 81a] [Apollo 81b] [Nelson 83] provides general-purpose sharing capabilities for local networks of personal workstations. The system is designed as a base level operating system, but can accommodate various machine hardware architectures.

The Apollo Domain provides a global namespace that spans the entire distribution of files across the network. File names are hierarchical and include the name of the node on which they reside. Users share single copies of programs and data files (i.e., the system does not support object replication) through a network-wide virtual memory system. Remote files are mapped from their physical address space to the virtual address space of the

processor that is requesting them. The actual data is then transferred as a result of demand-paging.

The Apollo Domain user environment is further shaped by the hardware environment. A bit-map display terminal is associated with each user workstation. A user can divide the display into multiple windows and execute an independent program in each, switching among them at will. This ability for concurrent activity on the part of the user and the system increases productivity and the effective use of the network environment.

1.5.2.2 Rochester's Intelligent Gateway

Rochester's Intelligent Gateway (RIG) [Lantz 80] [Lantz 82] is the University of Rochester's distributed system composed of heterogeneous hosts interconnected via multiple networks. The system has been evolving since 1974. It is implemented at base level on the gateway machines, and as guest-level extensions to other hosts. RIG was designed to provide access to all resources on the network from a single terminal. An equal amount of attention was paid to the end-user and program interfaces.

The RIG file system is visibly distributed in that it has a hierarchical namespace extended to include host name. Users and programs can access local and remote files with the same semantics. There is no central catalog in RIG, and replicated files are not permitted.

RIG allows users to access existing services or tools through three mechanisms. All available remote tools can be accessed using a virtual terminal protocol to attach the user's terminal to the appropriate remote host, then accessing the tool with the conventions of that host. For certain hosts, RIG provides access service analogous to that provided by Network Access Machines. The highest level of service is provided for tools that have been installed into RIG. Installed or "encapsulated" tools (an extendable set) are separated into tool interface and tool service components. While the service component always executes on its own (possibly remote) host, the tool interface component can run on the same host as the user to ensure fast response. In addition, users may tailor their own command interfaces for frequently-used tools.

Concurrent work is allowed and encouraged by RIG. Because of the RIG Virtual Terminal Management System, users can allocate separate windows of their display terminal to separate independent jobs. A complete virtual terminal is available in every window of the screen.

RIG's software architecture follows the "process" system model. The world consists of "resources" such as files and devices, that are managed by "servers" and can be accessed by "clients". Each server deals with its resources in its own way, and provides a message interface to other processes. Clients and servers communicate by passing messages.

In RIG, reliability is based solely on mutual suspicion; there is no replication of databases or control functions. Resource management is decentralized, with user and system resource selection. Administration too is decentralized, with each RIG host being managed by a logically independent organization.

1.5.3 Sharing Among Heterogeneous Hosts

There are a few systems discussed here that are explicitly designed to facilitate sharing among heterogeneous hosts. These include NSW, Desperanto and CRONUS, and LINCS.

1.5.3.1 The National Software Works

The National Software Works (NSW) [Millstein 77] provides terminal users, programmers and project managers uniform access to tools or services on different heterogeneous hosts. The system was designed and implemented during the mid- and-late 1970's as a joint effort on the part of Bolt, Beranek and Newman, Inc., General Systems Group, Inc., Honeywell, Massachusetts Computer Associates, Inc., MIT, SRI International and UCLA. The project was supported by the Department of Defense through ARPA and the Rome Air Development Center. Its initial implementation is on a collection of heterogeneous hosts on the Arpanet, including machines running the TOPS-20, IBM MVS, MULTICS and UNIX operating systems.

NSW provides its users with a unified tool kit distributed over many hosts, and a monitor that provides

- a uniform command language to all tools,
- a global file system, and
- a single access control, accounting and auditing mechanism.

NSW supports existing tools (those developed for single-site use) that have been installed into the NSW toolkit. A single NSW account permits access to installed tools on all NSW hosts, sparing the user from having to obtain login accounts on multiple hosts. After logging into NSW, either from a participating host or directly from an Arpanet TAC, a user can access tools on different hosts without needing to know about various host operating systems with their differing command languages, file accessing conventions and login procedures; knowledge of how to use the tools needed for a particular job suffice. NSW supports the coordinated use of several tools (where the output of one tool may be used as input to the next) by providing transparent file transmission and reformatting (translation of data from the representation of the sender to the representation of the receiver).

The NSW File System provides a central catalog of objects. Objects consist of files and services. Object names are location-independent and non-hierarchical. Access permission is associated with the user rather than with the object to be accessed. The NSW file system is not based on dedicated on-line storage, but instead uses facilities contributed by various host operating systems. Users need not be familiar with file access conventions on these storage hosts.

The NSW system itself is implemented as a collection of processes or components, distributed across the network, which cooperate to provide NSW services. Coordination is provided by the NSW monitor, called the Works Manager, which at least conceptually can be distributed as well. NSW is built on its own interprocess communication facility (MSG); this eases its implementation on a variety of network systems. Each host in NSW

has an MSG-server process. The user interface is provided by an NSW Front End component; there is one incarnation for each active user. A Foreman process is attached to each active tool and provides a tool execution environment. One of its objectives is to ensure that the tool refers to global NSW objects instead of objects local to the tool (except, of course, for files which are adjuncts to the tool's code, such as help files!). Each host providing NSW file storage has a File Package to handle cross-host file transfers.

1.5.3.2 Desperanto

Desperanto [Mamrak 82a] [Mamrak 82b] [Mamrak 82c] [Mamrak 83] is a network operating system that has been under development at Ohio State University since fall, 1980. It serves as a vehicle for the research general-purpose systems supporting resource sharing in networks of heterogeneous hosts. One of the goals of Desperanto is that solutions be generally applicable and require no changes to local operating systems or to the code that implements the sharable resources.

Desperanto provides a consistent and coherent view of the network environment to end-users, and a set of common network services to application programs. Users log into Desperanto from their local host operating systems to access tool modules that have been installed into the Desperanto tool bag. Application programmers can tailor "module interfaces" to integrate modules that have been developed for uniprocessors into the distributed environment. A module interface specifies those operations which the module provides to the system, those operations the module requires of the system, and other module-dependent information such as data types, conversion information and exception-handling routines.

To aid the installation of existing modules Desperanto supports a synchronous remote procedure call IPC facility. When such a module attempts to call on an operation provided by a remote module, Desperanto intervenes and uses the remote procedure call mechanism to make it appear that the operation was performed locally. In addition, Desperanto supports a message-passing view of interprocess communication for new modules developed to take avantage of the distributed nature of the system.

The Desperanto system relies on the work of others to provide a distributed file system and bases itself upon the LINCS transport layer (discussed in [Fletcher 82]). There are three primary areas of research being pursued by the Desperanto project, namely:

1. Guest Layering on Local Operating Systems

2. The Tool Installation Problem

3. Developing Distributed Applications

The first of these areas is concerned with the layering of distributed systems on top of heterogeneous local host operating systems. These guest systems must rely on local architectures to provide such primitive services as local and remote interprocess communication. The work here centers around the development of a virtual interface for local IPC, and of a separate virtual interface for remote IPC.

The second area is concerned with solving the problem of supporting existing single-site tools in a distributed environment. References to names of remote objects must be trapped by Desperanto during tool operation so that objects may be made local to the tool in a manner that is transparent to both users and the tool. Complexity is added when both users and tools can name objects during tool execution and when tools allow for dynamic naming of objects during their execution. The system must provide facilities for the conversion of data from the internal representation of one host to that of another when pipelined tools reside on heterogeneous architectures. This conversion is of high-level data structures such as arrays, records, tables and trees, and of well-understood low-level objects like characters and integers. Research into the exception-handling facilities needed in a distributed environment is being done, and into ways of providing those facilities, is being carried out as well.

The project team has also been designing new distributed applications to test the facilities of the Desperanto system. These currently include a distributed measurement facility and a calendar scheduling and maintenance system.

1.5.3.3 Cronus

Cronus [Schantz 82] [Hoffman 82] [Schantz 83] [MacGregor 82] is a general purpose system designed by Bolt, Beranek and Newman, Inc. to facilitate resource sharing among a "cluster" of nodes on a local area network(LAN). The Cronus cluster model includes a communication subsystem that interconnects two types of nodes: Cronus cluster hosts ("Generic Computing Elements", or "GCEs", which are dedicated to Cronus functions), and application hosts. Application hosts, which may be heterogeneous, require some operating system enhancements, and can participate in Cronus with a variable degree of integration. The Cronus system can be accessed from a variety of points including personal workstations, general-purpose mainframe computers, and the internet. GCEs share a common architecture and thus provide a replicated resource, i.e., Cronus system modules can be replicated on several nodes for increased reliability and availability. Cronus is based on a virtual local network (VLN) that stands in place of a direct interface to the local area network. This feature allows implementation on various local network architectures with relative ease. I/O services are provided to permit the exchange of data between distinct Cronus clusters, and between a Cronus cluster and the internet.

From the end-user's point of view, Cronus provides a single account and uniform access to all integrated system services, a distributed file system and a coherent execution environment. Cronus differs from NSW and Desperanto in that it supports only those applications written specifically for use with Cronus; existing pre-Cronus software is not supported. The system provides facilities for data translation between heterogeneous hosts, and supports interactive access to remote programs and multi-host pipeline processing (the chaining together of tool executions, such that the output of one tool becomes the input of the next). Cronus incorporates its own electronic mail system. End-users can work on several tasks at a time if they use Cronus-supported multi-window CRT terminals.

Cronus presents an object-oriented model to end-users and programs. Object management (cataloging and manipulation) is provided for Cronus objects such as users, directories, files, programs and devices. Objects are recorded in the Cronus Symbolic Catalog, which implements a cluster-wide hierarchical location-independent namespace. The catalog maps user-oriented

symbolic names into program-oriented identifiers. Cluster-wide user identifiers and user authentication forms the basis for uniform access control to all Cronus resources.

The Cronus File System supports several (including replicated) file types, and allows Cronus files to be linked to each other or to external (non-Cronus) files.

1.5.3.4 LINCS

Richard W. Watson and John G. Fletcher at the Lawrence Livermore Laboratory have proposed a comprehensive network architecture designed to support effective resource sharing through the development of distributed programs. This architecture is called LINCS (Livermore Interactive Network Communication System) [Fletcher 82] [Watson 82] [Watson 83].

The LINCS architecture is capable of efficient implementation as a base (native) operating system or as a guest layer on top of existing operating systems that support adequate interprocess communication. Systems implementing LINCS as a guest service allow access to resources from both their native and LINCS environments.

The network operating system model presented by Watson and Fletcher provides a coherent view of distributed resources to processes, programmers and terminal users. Network and location transparency is emphasized, so local and remote resources are accessed identically in application programs. It is, however, possible to discover the location of an executing program, and for a programmer to influence where a particular program should execute or from where a resource is to be obtained. Later, when the issues are better understood, automatic allocation and location on a global level will be added to the system. The development of distributed applications is further facilitated by the abstraction of common network-interaction issues into well-defined conventions and protocols.

The NOS system model appears to be a hybrid of the object and process models. Objects such as processes, files, directories, devices, etc. are called 'resources'. Resources are managed by

'server' modules which maintain the abstract representation of the resource, and implement allowable operations on the representation. Interprocess communication is message-oriented. A given process can act as a 'customer' or server of a resource at different times (so that a server of one resource can be a customer of other resources).

A process accesses a resource by sending 'requests' containing operation specification and parameters to the appropriate server. The server then satisfies requests by accessing data structures local to it or by sending requests to other servers for help in servicing the original request. When a request is satisfied, the server sends 'replies' containing (a) indication of success or failure, and (b) results if any. LINCS supports efficient transaction (minimum delay and message exchange) and session (extended conversation) communication modes. Synchronization is provided in the form of blocking and non-blocking sends and receives.

The LINCS system supports two levels of resource names. High-level human-oriented pathnames are implemented by directories and directory servers. Low-level machine-oriented names are "capabilities". In addition to naming resources, capabilities are used to validate access to resources, to identify specific transactions uniquely, and to indicate sources and sinks of information, as well as places to which replies are to be sent. Resources are shared by exchanging capabilities in messages.

Capabilities are used for protection, but the capabilities themselves must be protected, since no asssumptions can be made about the ability of the participating autonomous host operating systems to guard against attempts to forge or insert stolen capabilities into the system. In the LINCS NOS, capabilities are stored in user-process space and protected by embedded passwords and encryption. In addition, the system supports controlled and uncontrolled capabilities; a resource server can choose to accept either or both types.

The LINCS architecture supports NOS-wide error recovery at the IPC level.

1.5.4 Communication-Oriented Systems

Presented here are two operating system kernels organized around the abstraction of communication. They form a basis for higher level services provided by a network operating system.

1.5.4.1 TRIX

TRIX [Ward 80] is a communication-oriented kernel operating system designed at MIT for use on a network of interconnected homogeneous processors. It is a generalization of many important features of the UNIX and MULTICS operating systems, including hierarchical directories, pipes and virtual devices. The functions provided by TRIX are a superset of those provided by UNIX.

The TRIX system semantics revolve around a stream communication mechanism rather than around passive objects like files. The two fundamental elements in the TRIX system model are processes and streams. A process comprises active computation and state information. A stream is a full-duplex communication path between processes, over which pass uninterpreted data and control messages.

Despite their full-duplex nature, streams are asymmetric, having a "handler" end and one or more "requestor" ends. Streams incorporate a capability-like mechanism protected by the system. Capabilities are passed between processes via "request" and "reply" messages. Communication along streams is asynchronous and non-blocking, so a process can have several active transactions at once.

System objects like files and devices are implemented as processes that react to traditional control messages (received on attached streams) in conventional ways and respond by sending control and/or data messag[illegible]ong different attached streams. The important point is that the semantics are associated with the streams, independent of the attached processes which implement them. Any system object can be transparently replaced by any other that mimics its communication patterns. Thus TRIX allows uniform and transparent access to local and remote objects.

TRIX names are hierarchical, and associated with streams rather than with objects or processes. Directories are processes that perform the name-to-stream mapping.

The TRIX approach to the implementation of passive objects like directories and files as active processes results in serious efficiency problems. TRIX performance is speeded up through the implementation of common system functions as highly efficient processes. The untyped nature of most of the data communicated over streams prevents the TRIX system from being implementable on a collection of heterogeneous hosts.

1.5.4.2 ACCENT

ACCENT [Rashid 81] is a communication-oriented operating system kernel developed at Carnegie-Mellon University as the basis for two quite different distributed computing projects. ACCENT supports a loosely-connected collection of host machines. Each host on the network runs the ACCENT kernel which provides the following functions:

- Interprocess Communication
- Virtual memory management
- Process management
- Process creation and deletion
- Access to devices through IPC
- Support for language and application-dependent microcode
- Rudimentary support for process monitoring and debugging

The system can be viewed as composed of a number of layers. The bottom layer is the kernel, and each successive layer builds more and more comprehensive services upon the next lower layer. Layers are glued together through interprocess communication.

In ACCENT IPC is message-oriented; its basic abstraction is the "port", a protected kernel object. Ports cannot be directly manipulated by processes; rather, a process requests "capabilities" from the kernel to send messages to a port and/or extract messages from it. The capability is a local name for a system object, much like a file descriptor in conventional operating systems. Ownership and access to a port can be passed in messages, but not shared.

Communication is extended transparently to remote hosts by network server processes. The location of a port can change, as can the body of the process serving it, without affecting customer processes using it, so long as interface arguments are maintained. This allows transparent process migration.

Communication speeds in ACCENT are much better than those found in many communication-oriented kernels. Virtual memory, file storage and interprocess communication are integrated in such a way as to provide IPC through cross-network paging.

1.5.5 Integrated Systems

1.5.5.1 Eden

Eden [Lazowska 81] is an integrated distributed system being developed at the University of Washington. It aims to combine the benefits of distribution and integration by using local area network technology and an object-oriented software environment. Eden is intended to run on a short-haul network of homogeneous personal computers.

Its current hardware architecture consists of a number of node machines (based on the Intel iAPX processor) with bit-map to display terminals interconnected by an Ethernet local area network. Eden allows various configurations of the basic node type, but does not in general support heterogeneous nodes. Foreign machines can be interfaced to the system through Eden nodes, however. In this case users on Eden nodes can access services on the attached foreign node, but the reverse is not possible.

Eden presents an object-based user and programming environment. An object encapsulates a resource, and consists of a unique name, a representation (data part) and a type, which defines the set of operations that may be invoked on the object to manipulate its representation. Objects (including users) access other objects using capabilities, which contain the name of the target object and access rights. The Eden user sees a location-dependent address space; it is the responsibility of the Eden kernel to resolve object names into location- dependent addresses and to forward invocation messages to objects whenever referenced. The semantics of object invocation are those of a blocking procedure call: the user or invoking process suspends execution until the requested operation is complete. A user may specify in the invocation a timeout period if he wishes to be notified when invocation is not completed within the specified period of time. Asynchronous execution (non-blocking invocations) are also supported.

The internal semantics of Eden objects (dealt with by the Eden system programmer) are much more complex than the external (user) view of these objects. Though to the user Eden provides a location-independent address-space, in reality location may be critical to system performance or reliability. The details of object location, as well as those of concurrency and error recovery are encapsulated within each Eden object. In Eden, objects are capable of obtaining information about their location from the kernel, of making location changes, and of being replicated and cached at multiple sites to increase performance, availability and reliability.

1.5.5.2 LOCUS

LOCUS [Popek 81] is UCLA's network-transparent high-reliability, high-performance distributed operating system. A prototype implementation is built on a number of PDP-11s connected by a variety of Ethernet-like local area networks. The component machine architectures vary widely in processor power and storage capacity; in fact, LOCUS can accommodate processors with no local file storage at all.

Each LOCUS host exercises a fair degree of autonomy. It is master of its own resources and can operate gracefully alone. To a great extent maintenance of the internal consistency at any

given machine does not depend on the correct operation of any other site in the network.

LOCUS is an integrated system, in the sense that each machine runs the same software. Each is a complete facility with a general file system, a name interpretation facility, etc. The LOCUS operating system is application-code compatible with the UNIX operating system.

LOCUS is a network-transparent system, in that all objects are accessed in an identical manner, without regard to object location. LOCUS supports a network-wide location-independent naming structure. End-users and programmers refer to LOCUS objects with high-level names. These names are globally unique and form a single uniform naming tree, in which each object is identified by its pathname. There is no notion of object location in these pathnames. Low-level names are also globally unique and can accommodate replicated files.

The LOCUS claim for high-reliability and availability is supported by the system's facilities for object replication and atomic update. LOCUS automatically replicates resources to the degree indicated in associated reliability profiles. Copies of a replicated file can be substituted for one another with no visibility to users or application programs. LOCUS supports file committing, such that for a given file and a set of transactions against that file, one can be sure that either all of the updates are done, or none of them are done, even in the case of network or mode failures. Updates are propagated to other sites by demand-paging. When a site is disconnected from the network, it can still process local work. Furthermore, when the network is completely partitioned, and copies of a replicated resource are found in more than one partition, the resource can be modified at the various partitions. These capabilities are made possible by a centralized LOCUS synchronization mechanism with distributed recovery.

LOCUS provides a level of system performance that compares favorably to stand-alone UNIX. Several design decisions led to such good performance.

- Specialized problem-oriented protocols were designed to handle common operating system functions (e.g., reading a file), and expensive redundant low-level network protocols were dropped.

- A very fast process mechanism for serving network requests was implemented inside the operating system kernel.

- Special handling is provided for local operations, so that network support is sidestepped.

- Lastly, the integrated system design speeds up processing since operating system functions are available locally to every executing process, no matter where it resides.

2. On the Utility of Distributed Systems -- by Charles Muntz

2.1 Rationale of the User Model

The National Software Works (NSW) is a comprehensive effort aimed towards "...providing programmers access to tools on different hosts...". These tools are essential in their "...attack on the cost and complexity of developing and maintaining software...". The difficulty in managing programming "...lies not in the [absence] of suitable tools, but in their [non-]availability...". The overall objectives of the NSW, then, "...are to provide programmers with a

- Unified tool kit -- distributed over many hosts -- and a
- Single monitor with
    * uniform command language,
    * global file system,
    * single access control, accounting, and auditing mechanism."

The above quotations from a recent NSW report describe user needs as perceived in the mid-1970's. There has been considerable technological evolution since this original problem statement. A review of the NSW experience should therefore begin by examining that statement for relevance to contemporary issues, since significant technical issues have been solved by developments in somewhat unrelated areas (just as the need to optimize program performance on a computer with drum main memory was obviated by the introduction of random access memory, and software constructs for overlaying programs and for managing large data structures in a relatively small amount of real memory became less critical with the advent of virtual memory environments). So, is the problem which NSW addressed still current?

The theme of the original problem statement is provision of

integrated tools as the key to reducing costs and coping with the complexity of the software milieu. The emergence of UNIX(tm) as a major computing environment testifies to the acceptance of tools and illustrates the utility of their integration. ARPAnet hosts offer powerful tools too, but they are difficult to access, and little attempt has been made to integrate them.

Our experience with use of the NSW system indicates that the audience for tools is much wider than suspected. Indeed, it should be argued that anyone whose productivity is enhanced by access to computing assistance should be included. Just as the implementor needs a superior code analyzer and his manager a good configuration tool, so also does the circuit designer produce more with the right CAD/CAM facilities, the technical writer with proper documentation tools, and the program manager with good tools for forecasting personnel availability. These latter examples belong in the category of applications software, a field which has experienced considerable expansion over the past ten years. We will not attempt to change the meaning of the word "tool" to include applications software; instead we will label as a "service" any sort of productivity-enhancing computing assistance. The first step in revising the NSW problem statement is to focus on providing a unified kit of services. We seek a unification of distributed services of the kind exhibited by UNIX tools when the output of one is piped to the input of another.

The NSW report next asks how a project can obtain appropriate tools. "If some essential tool does not happen to have a version which runs on a computer to which the project has access, the manager is forced to choose among the expensive alternatives of (1) foregoing use of the tool, (2) undertaking to acquire or produce a surrogate tool on his hardware, or (3) purchasing [access to] a computing system on which the tool does run." The tool of dubious utility would probably be abandoned. A valuable one might be ported to one of the project's machines, if this could be done in a technically acceptable and economical fashion. "NSW tries, in effect, to make alternative (3) more attractive..."

One technological event could render this reasoning obsolete: the emergence of a single computing architecture capable of accommodating the myriad of systems available today. That architecture would necessarily be amenable to excellent

implementations of systems for data processing, scientific computing, interactive environments, etc. If one could build such a machine at all, the problem of providing a unified kit of services would be vastly simpler than the NSW problem statement envisions. An homogeneous network would be sufficient to link enough processors together to do almost any job. If one could also build such a machine to any desired scale, all services could run on the same host. Such a breakthrough would offer one (indirect) solution to the original problem.

Today's computing is done on machines of disparate architecture. Seeking the best in data processing often leads to machines of one manufacturer, whereas seeking the best in interactive environments, in office automation systems, and in scientific computing often leads to machines of three other origins. The architecture and operating systems of these machines are as specialized as the problems which they address. Announcements of new architectures are appearing at an increasing rate: data collection systems, graphics systems, and back-end database machines to name a few. This trend will continue. Increasingly specialized machines will also be introduced, thanks to the twin economic factors of decreasing hardware costs and increasing productivity in design and manufacturing attendant in CAD/CAM technology. Convergence on an ultimate system architecture therefore seems highly unlikely. The unified kit of services should be expected to require hosts of differing architecture; so the solution to the problem addressed by NSW necessitates distributed processing on non-homogeneous hosts, as before.

User access to the heterogeneous network is a challenging problem. If users have terminals which can be used interchangably on the network's hosts, the problem is reduced to one of establishing and maintaining terminal connections to these hosts. However, a significant portion of the diversity in computing systems lies in their terminal-handling disciplines. Systems which excel in forms management are based on screen-level operations; on the other hand, highly interactive systems involve their mainframe in the processing of each keystroke. Since many desirable services have such rich interfaces with the user's terminal, access to a terminal familiar to the tool is prerequisite to satisfactory usage. Another step in revising the NSW concept is then to widen the view of devices through which users access the system, and to examine carefully the match

between the user interface and the complement of services to which he wishes access.

Since a heterogeneous network is the only system capable of offering a unified kit of services, we need to examine requirements for an operating system for the extended computing environment. Without a system such as NSW, a user could try "using [for instance] the Arpanet in straightforward fashion, by using Telnet and FTP to access hosts other than one's 'home' system, which does indeed give you a much wider domain of action, but nonetheless:

- You need an account on each host...
- The operating system on each host is different, so you must learn different login procedures, command languages, interrupt characters, file naming conventions, etc...
- Files output from one tool...are to be input to another tool... This involves at least network transmission and usually file reformatting..."

The burden this places on training, document acquisition, and accounting, as well as the overhead required for managing non-essential bulk data transport, makes this solution very costly.

As we noted earlier, another concern should be cited here: the degree to which the local host's usual terminal is appropriate in heterogeneous environments. Use of the word "host" now appears to have the undesirable effect of suggesting that system mainframes define user interfaces. In fact, user interface hardware is of significant importance in shaping the operating environment in which services are offered. Since NSW focused on the problem of accessing tools, we wish to adopt the viewpoint that service delivery is accomplished through cooperation of a trio: the mainframe, the terminal, and the operating system. We call this complement the operating environment of the service, and therefore revise the NSW statement further: to provide a unified kit of services distributed over heterogeneous operating environments.

The introduction to the NSW report then concludes: "The purpose of NSW is to make [a unified tool kit -- distributed over many hosts] a practical reality. The NSW user should not have to know about OS/360, TOPS20, and MULTICS with their differing file systems, login procedures, system commands, etc.; knowledge of how to use the individual tools which are needed for the job should suffice. He should not have to worry about reformatting and moving files from a 360 to a TOPS20; file transmission should be completely transparent. The user should not have to worry about obtaining accounts on many different machines, but instead should have a single NSW account."

Transparency is indeed a crucial consideration. Suppose one wishes to run a tool on MULTICS over a file of information. In the context of heterogeneous network operating systems, the input file may be stored on a variety of hosts. The network operating system will be transparent if and only if the identity of the file's storage host is invisible to the human observer (without examining internal system information, of course). In other words, transparency has not been achieved if the user must be concerned about whether a file is stored

- local to the tool, or remotely
- on some machine running a particular operating system
- on a specific host
- ...

Transparency certainly demands absence of distortion. If we were to attach the same printer to every machine in a network exhibiting the desired property and then list some text file everywhere, the resulting stacks of paper should be indistinguishable. There are other attributes which demand equivalence, however - most notably the comparative dynamics of tool operation. In our NSW prototype, the fetching of remote files was visibly slow; even though the fidelity of file translation was relatively good, we did not in fact achieve transparency.

Another barrier to transparency is raised by the multiplicity

of failure modes that may occur in a distributed environment. If the file to be input to a tool resides on hosts which are down, no obvious method is available to the tool to report such a fact to the user of the tool. (A good analog is the case in which the desired file is located on a dismountable disk which is not available when the file is requested, but the problem remains: transparency is lost because error conditions are not equivalent.)

We can, however, hope to achieve an acceptable equivalence: normal operations of service behavior do not reveal the intermediary network, and the extended faults that do occur are reported in the same fashion as similar system errors, etc.

In conclusion, the problem NSW originally addressed is indeed current, and the need for a system of this class has increased over the decade during which NSW was conceived and developed as a prototypical evolutionary step towards a distributed heterogeneous operating system.

2.2 Characteristics of a Distributed System Supporting Maximal Access to Products

Bonnie Shipman's "Survey of Distributed Systems" listed a number of distinguishing characteristics. As we look towards the next generation system, we next wish to consider each one of those characteristics -- their criteria together with our recommendations and supporting arguments -- as influenced by our experience with the NSW prototype. We call this system MAP: Maximal Access to Products.

1. General Purpose Vs. Special Purpose

   MAP inherits generality from the products it offers; thus we would expect general purpose support to be provided. It is important to note that nodes in a MAP network need not offer identical services; rather each node need only implement sufficient functionality to support locally-offered products.

2. Guest Level Vs. Base Level Implementation

MAP will offer a large number of products. Each such offering represents a substantial investment on the part of its builders. Products which can run in the MAP system without modification accrue a number of savings; their component parts -- software, documentation, training, customer support, etc. -- can be used without modification. Sophisticated products are themselves sophisticated users of their own operating systems. Since replacing the product's operating system appears to be prohibitively expensive, guest level implementation is clearly indicated.

3. Visibility of Distribution

Distribution in MAP will be observable by its users. Earlier, we discussed transparency as a goal and noted that, in all likelihood, products running in the MAP environment will be perceptibly slower than their centralized counterparts. In addition, certain components of a distributed system can fail independently of others; consequently, users have a richer failure model with which to contend. For example, a user running an interactive tool on a remote host who wants to process a file on yet another host will have to be cognizant of the effects of any combination of the three hosts going down. The advantages of a distributed system will motivate the sophisticated user to deal with these problems. User comprehension may well go hand-in-hand with high-level fault isolation; a decision to report system errors in this manner would indeed make distribution in the system clearly visible.

Should host identity be revealed? It is attractive to hide this information from the users so that tool availability could be reassigned among compatible hosts by network administrators. During our experience with NSW, however, hosts were never interchangable; even if they ran the same operating system, each host's tools inevitably acquired a distinct personality. Thus, products in MAP will be identified as to their host of residence.

4. Network Services Provided

This issue depends upon the position taken on visibility. If distribution is to be visible to the MAP user (and we believe it must be), then user-level commands reflecting the status of the distributed system will naturally appear. In NSW we took the position that our users' access module (the Front End) would supply non-NSW network services -- two important areas of functionality included access to external file systems ("import" and "export") and establishment of terminal connections (ARPAnet TELNET).

5. Reliability

A distributed system may be highly reliable, but only if its control functions are dependable. A system like MAP offers a multiplicity of resources: access points, file servers, and computing resources. Simultaneous failure of such resources is unlikely. Since operation of the entire computing network depends upon the availability of system control functions, our NSW prototype required availability of the Works Manager's host. Proposals to distribute the Works Manager were never pursued, due largely to our estimation that system response would suffer dramatically if the control functions (and database!) were distributed. The development of a system architecture capable of simultaneously achieving desired performance and reliability is a necessary prolog to the design of MAP.

6. Protection

In line with the issue of system generality, MAP inherits protection requirements from the products it offers. But generally speaking, the products are not directly concerned with details of host system protection; thus MAP implementers need only address the issue of protection and sharing among network and local users. The unusual product which does include manipulation of local protection mechanisms could be difficult to offer in the MAP environment. For example, should MAP choose to implement user-based permissions, products which manipulate object-based permissions could not then be supported with any obvious strategy. Nevertheless, even though NSW employed user-based object permissions, it also

offered an acceptable variety of tools on MULTICS, which exerts control on an object basis. This is due to the fact that most tools do not concern themselves with the details of protection; rather these details are delegated to an -- any -- operating system. On the other hand, a sophisticated file managing tool (e.g. a code librarian) might prove difficult to offer.

Changes in the protection status of resources and users, generally performed by system administrators must, in a distributed system, be handled in a manner addressing the needs of distributed administration (see 11.).

7. Resource Selection

Two kinds of resources whose selection is of interest are the actual host for running a desired product, and the actual copy of a file to be used when the file system is distributed across heterogeneous hosts. If the MAP network offers installations of a given product on a number of hosts, then a specific host must be chosen in order to run that product. The choice can be narrowed -- if not actually made -- by administrative restriction. Algorithmic considerations for making the actual choice might start with preference towards the host serving as the user's terminal handler, since interaction should be optimal when the terminal is local to the product being run. Failing this, it is highly desirable to choose a host on which the product's input files are stored, obviating the need for file transmission and possible translation. Indeed, when the size of the input files is so great that it precludes relocation -- for example, a large data base -- the issue is decided . Finally, if separate hosts must be used for access, execution, and file storage, then file content translation need not be performed if executing and file storage hosts can be found whose operating systems are compatible. (The chosen host must of course be up!)

Our initial approach in NSW was to treat remote installations of the same product as equivalent. A user could type 'Use TECO', but was not allowed to

influence the choice of which TECO was actually selected, nor was he informed of the host's identity. While this system is ideal for network administration, it proved confusing to users. As we noted during our discussion on visibility, multiple tool instances were in fact not equivalent due to the variety of installation decisions at participating sites. Note that the NSW command language statement which named the tool did not specify the files on which the user intended the tool to operate, so that NSW could not exploit the desirability of co-locating the tool along with its files.

8. Resource Management

NSW used a central resource catalog containing names of users, tools, and files, along with user-based permissions for tools and files. In fact, NSW was vulnerable to failures at the Works Manager site (where that data was stored). Performance often suffered because of the interhost communication required for each access check or login. An exploratory form of distributed control was implemented: the TOPS-20 implementation of the Foreman (tool execution supervisor) maintained locally a private database of files which had been read and created during the tool execution session. If the central site failed during the session, or if the tool session aborted prematurely, the database sufficed to recover the files which were created during the aborted session. In fact, the TOPS-20 Foreman was far less dependent than other Foreman implementations upon the Works Manager; its performance was superior since it had less reason to contact remote resource managers. It was also the most complex area of the system to build and operate. When the Works Manager's records of tool sessions disagreed with the Foreman's records, system behavior became erratic. Intervention by system operators was generally required to re-synchronize the two databases. Certainly, improved techniques for system design and testing must be developed before ambitious functionality of this type can appear in a network product.

9. Extensibility

A network like MAP is responsive to needs for extension along several axes. Since MAP will offer an abundance of products, system features can - and should - facilitate additions in this area. New products which are offered in one of the operating environments which MAP already supports can be incorporated with little or no difficulty. In order to support the desire to add products on a continuing basis, however, MAP should be extended to new operating environments in response to demands for additional products. The original NSW prototype offered tools on TENEX, OS/360 MVT, and MULTICS. As the project progressed, TOPS-20 (as a variant of TENEX) and MVS (as a variant of OS/360 MVT) were added, and a UNIVAC EXEC8 implementation was designed.

While product addition is an example of pure extension, evolution of the system represents a related need. Thus, MAP's architects should plan for the replacement of critical elements with improved variants of the originals, thereby incorporating experience gained with earlier releases. An important arena in network evolution is the user interface. The NSW Front End (FE) experienced more architectural variation than any other system component. As originally conceived by SRI, the FE included a repository for tool functionality concerned with user interaction, so that interactive tools would become distributed tools as well. A TOPS-20 FE was developed by COMPASS chiefly in support of NSW evolution; it had to be satisfactory to use and capable of supporting all aspects of NSW development, but it was never intended to provide full support to users. BBN designed and built a complete FE which could be run as a UNIX tool. The UNIX FE contained support for running programs as sub-processes, for listing files, for defining keyboard macros, for connecting to other network systems, etc. Even though each of the three FE's presented a distinct user interface, they used a common interface to remaining system components; no dependence on the FE type appeared outside the FE.

10. Administration

The choice between two distinctly different forms of network administration exerts significant technical

influence on the system itself. At one extreme one finds interdependence: nodes as embodied in the Apollo Domain architecture. In this freely distributed file system any user's files can physically reside on any other user's disk. Thus, while Domain users may choose to dedicate nodes to selected users (placing a workstation in a user's office, for example), only the display system is subject to exclusivity. The Domain operating system may choose to allocate files on any given disk; it may request cycles on storage nodes' CPUs, etc. In fact, no user may power down a node: the on/off button has no effect on hardware required for remote file services.

Federation is the usual administrative relationship among the nodes of a network. Even though Domain is composed of distributed computing elements, the administration of these elements is centralized; when administration is also distributed, a different system model is required. It cannot be assumed that remote resources will be available whenever local users need service, because remote hosts may have disabled network service on the basis of local priorities. Our NSW nodes resided on university machines (on which end-of-semester deadlines reduce system resources); and on government-operated machines (which are sometimes commandeered for priority demonstrations of other projects). Such perturbations fortunately yield to system adaptation mechanisms. They can therefore be viewed as another source of unreliability - a node can fail for administrative as well as for technical reasons. Since MAP plans to add computing resources over time, its architects should assume the conservative position of the federation model.

Another aspect of system administration is establishing and maintaining user accounts and their attendant permissions. Most organizational methods for performing these tasks rely on the rich communication channels which one finds in interpersonal cooperation: face-to-face discussions, telephone calls, etc. When the user community is widely distributed, communication between system users and administrators becomes limited, and cooperation is harder to achieve. Indeed, one may have to rely on electronic mail to accomplish tasks rather than as a

mere supplement. NSW developers had a great deal of trouble with these communication limits. (Complex but typical examples might be helpful. An electronic message is sent to persons at different locations. If the manner in which the sites cooperate in reacting to the message is unclear, there is no means within the message of establishing cooperation. If someone received a message describing a system bug, was it just for information, or was the recipient thereby requested to fix it?) In response to such difficulties, a structured "mail" system was developed to coordinate the efforts of the distributed project staff. Such a tool is clearly needed in MAP to facilitate communication between users and administrators.

2.3 Applications of a Distributed System Supporting Maximal Access to Products

In this section we turn our attention towards the utility of MAP, a system providing a unified kit of software products distributed over heterogeneous operating environments. MAP clearly has unusual characteristics -- most notably, the integration of multiple computer systems, even though they have disparate architecture and are geographically distributed.

How can MAP's distinctive features best be exploited? Such considerations are especially important for systems whose functionality reaches into the higher layers -- application and presentation -- of the ISO/OSI Reference Model. This certainly applies to MAP, which must effectively translate file and terminal characteristics. Based on our experience with NSW, the following examples illustrate the potential benefits which a system like MAP can realize.

1. Software Product Evaluation

   It will be easy to evaluate software products in the MAP environment. At first, one might think that the source of leverage in this area lies in the multiplicity of operating environments found in the mature MAP. As this number increases, so does the probability that new products can be installed in MAP

without (the relatively expensive task of) first installing its operating environment. MAP's real advantage lies elsewhere, however.

Products do not operate in isolation. Borrowing from John Donne, "No product is an island". Most such packages exploit the rich file systems in which they operate; furthermore, packages are often used in groups to form problem solutions cooperatively. A spelling checker provides a good example: user-written document files are scanned for spelling errors using standard and custom dictionaries. In order to evaluate the spelling checker, a user must create document files and custom dictionary files and finally, invoke the tool. The result of the check must in turn be processed: printed, interactively browsed, combined with other documents, etc.

It can be argued, therefore, that more effort must often be invested in acquiring a working knowledge of operating environments than in using the tool itself. In order to create and manipulate files, naming rules must be learned. To compose text, an appropriate terminal must be found and editors must be learned. To invoke the candidate product, execution rules must be learned along with attendant exception handling and reporting conventions. And finally, in order to share results with others, protection rules must be learned.

MAP's real advantage lies in its standard operating environment. In order to evaluate the same spelling checker within MAP, no new rules need be confronted; the file naming, execution, exception handling, and protection mechanisms are standard. More importantly, familiar editing and file manipulation tools can be used. It is not that editors are hard to learn; the problem lies rather in the difficulty (impossibility?) of switching freely among sophisticated editors with which one has acquired proficiency. Evaluation in MAP can thus begin with a concentrated appraisal of the product under evaluation, or in this example, "How do I create my custom dictionary?".

2. Evolution toward Distributed Applications

One interesting possibility for the MAP network is the

distributed application. While the previous discussion shows that new products can be easily investigated, MAP minimizes the learning requirements for such products without actually producing any new functionality. Suppose, however, that we want to share the results of an attractive product with others -- other persons and other products! MAP's power of integration is most clearly demonstrated when the participants in its scenario are distributed.

Its capacity for accessing an entire network via a single-step authentication is highly useful. Elementary file-sharing operations are far easier in the MAP network than in one consisting of a loose federation of hosts. As we stated in the Rationale of the User Model,

- "You need an account on each host...
- The operating system on each host is different, so you must learn different login procedures, command languages, interrupt characters, file naming conventions, etc..."

The NSW Technology Demonstration provides some good examples of this. Three Air Force Logistics Centers participated: Warner-Robins (GA), Oklahoma City (OK), and McClellan (CA). Each site has a PDP-11/UNIX Front-end Machine. Tools were offered on TOPS-20's in Rome NY and Los Angeles, on an IBM/MVS system in Los Angeles, and on a Honeywell/MULTICS system in Rome NY. All machines are connected via the ARPAnet, but each is separately administered. Using conventional ARPAnet protocols, each machine would have to be confronted anew in order to access a remote tool or file. Our users found NSW's central access to be much more convenient. Central authentication, then, is one demonstrated advantage of a network like MAP.

Realistic applications of tools (and products) involve a cascade of processing. To accommodate this, UNIX provides the ultimate environment: Standard Output from one tool can be internally "piped" to Standard Input of others, creating a tool complex appropriate

for a large range of problems. Another new functionality which a network like MAP makes possible, then, is the distributed complex of products.

The NSW Technology Demonstration included several scenarios for system usage. The most ambitious of these was the Emulation scenario, which exemplifies distributed applications of this type. The Nanodata QM-1 is a novel computing device which can assume the CPU architecture of a very wide range of machines. It accomplishes this by loading appropriate vertical and horizontal microcode -- DECsystem-10, IBM/360, and microcomputers are examples of machines which can be simulated. The QM-1 is usually attached to a general purpose machine in order to provide the customary array of support peripherals (and networking!).

Quite a lot of software is required in order to use the QM-1. For example, suppose we want to test a piece of avionics software as it might run on an Intel 8086. The first stage configures the QM-1 microcode so that the system will behave like an Intel 8086 CPU: its registers, addressing, timing, etc. We are then in a position to load the QM-1's "memory" with the binary data -- the results of compiling, assembling, and linking the avionics software -- which would be loaded into the 8086's memory. The QM-1 is now in a position to "play" Intel 8086 on the avionics code. The CPU can be started at selected locations, breakpoints can be inserted, etc.

The NSW Technology Demonstration provided a distributed complex of tools in answer to these many needs. A QM-1 was connected to RADC's TOPS-20. The SMITE compiler, which generates the QM-1 microcode from a high-level description of machine architecture, was installed on RADC's MULTICS. Cross-compilers and cross-assemblers were installed on these machines and on UCLA's MVS system. One of the participants had a QM-1 and could transfer remotely developed applications to the local machine. The other sites wanted to evaluate the QM-1, but did not yet have their own machines.

Distributed applications represent a new frontier of product development. Since no single system

architecture can satisfy user needs, groups of products which can fill the computing needs of today and tomorrow must be free to exploit the architectural diversity found in the heterogeneous network.

3. Configuration management - of distributed objects

Inconsistencies which come to light during product installation or upgrade imply the need for computerized configuration management. Consider some examples: the distribution kit for a product arrives with the user manual for the previous version; an upgrade kit includes only some of the product pieces actually affected by a change; testing a new product release did not consider regression in unchanged pieces. Such malfunctions of product management are not diseases, but rather are symptoms of an imbalance. What is the problem, and how can one treat it?

Each of these failures is a consequence of incompatibility among the product's component pieces. The goal of configuration management is to preclude such errors. Existing unautomated techniques suffer from an incomplete representation of the true composition of the product. A complete configuration includes documentation, test cases, desired and actual results of tests, training materials, etc. Only when the elements of this collection achieve coherence should the product be considered for distribution (i.e. for installation or upgrade).

Automating configuration management naturally begins by considering a database rich enough to represent product composition, along with comprehensive procedures to support product management -- installation and upgrade included. Such a database would have to represent the composition of sub-assemblies of the product. Procedures for managing the product might include composition reports, change impact analysis and notification, release generation, and trouble reporting.

Processing of this sort has been done for a long time: Bill of Materials applications were among the early commercial applications of computing. The configuration management problem could be viewed as a

variant of that problem. Its raw materials are the software, documentation of all kinds, test cases with actual results, etc.; its outputs include installation kits, upgrade kits, and reports.

In the "Rationale of the User Model", we discussed the continuing specialization of computer system architecture. This trend suggests that products of the future will be composed of pieces developed in different environments. In an advanced radar system, the hardware drawings could come from a CAD/CAM system, the embedded software from a programming environment, the documentation from a word-processing system, system simulation results from an array processor, and actual test results from data acquisition and analysis facilities at a testing laboratory. And the radar is only one sub-system of a ship, aircraft, etc. The management of an immense amount of information is difficult to visualize until we recall that current practice is to warehouse volumes of paper.

The global namespace of the MAP network allows construction and manipulation of a database which could accommodate a product of the complexity of the radar system. Furthermore, procedures to manipulate the named pieces could operate there. Thus, a network like MAP must form an implementation substrate for automating the configuration management of products whose pieces are taken from a heterogeneous enviroment.

4. Distribution - to dispersed consumers

The MAP network supports the production of documents and programs, and should in turn support their distribution to end users and to other producers. In a network environment, the scope of distribution can be very large. The interaction between remote producers and consumers requires special treatment, in the same spirit as does the area of distributed system administration [see characteristic number 9, above]. We assume here that the items being distributed may be revised from time to time.

Consider the production of something as simple as a

single document being written by a single author. During early stages of preparation, it is a private file accessed by the author only. At some point, the author may distribute working copies to others for comment. While the document at this point is no longer private, neither is it fully public (bibliographic references to such documents often call them "private communications"). Once the document is scheduled for release or ready to be used by others, a number of actions should take place to support both producer and consumer. Archival is generally performed on released files: consumers often demand it, and producers usually want their released items archived too. A release announcement might also be sent to an initial intended readership.

Once the document has been released, it can be accessed by many consumers; all one needs is the name of the file and read access to it. We now enter a period of increased perusal of the document, with (usually) less interaction between producer and consumer. A number of useful facilities could be provided to streamline access and to prepare for subsequent releases. The system could provide mail service if the producer's mail address is available, and (possibly) a forwarding service if not; a distribution log could be kept to show who received copies, and when.

Up to this point, nothing very "exciting" has happened. We have described normal preparation/distribution procedures. Now we get to the meat of the problem: authors may want to modify their documents. It is the consumer who is now faced with a number of problems:

- Can I find out when a change is released?
- Can I get update pages or a whole new document?
- How can I be sure I have the latest version?
- Can I recover any released version?

First, we must assign names to successive revisions. This is, in fact, one of the most appealing features of version numbers: succeeding revisions may be identical in name, but distinct in version number. Release of a change is signalled (to the system) when the producer releases a file whose previous version has already been released. If a distribution log were collected, change announcements could be sent to consumers (or at least to those who requested update notification when they obtained their copy). The change notice could include a summary of the modifications, as reported by the producer when he released the new version. Note also that update pages (the subject of the second question above) can easily be produced by processors like VMS DIFFER, TOPS-20 SRCCOM, etc. Finally, every version is separately archived, so that any particular revision is retrievable at will.

Sometimes one wishes the "latest" version of a document, and at other times a particular version is required. For example, if I am an author and need a reference for "current" usage, I might go to the bookstore and ask for the "latest" Webster's dictionary. In the book's bibliography, I would note that in fact the 1978 Webster's dictionary was used. It is most important that the specific revision be noted so that in the future (say the year 2200) the book could be read in the context of the proper dictionary. In NSW terms, I merely want the latest copy of WEBSTER.DICTIONARY when I "go to the bookstore", but I record the edition used (as WEBSTER.DICTIONARY;1978, for example) in the book's bibliography. If I do not wish to be notified of changes, I can always see which version is offered in response to WEBSTER.DICTIONARY. As long as it's 1978, I'm all right; if it changes to 1979, I can take a new copy if I wish.

A network like MAP provides integrated access across a broad geographic space. It is therefore the ideal medium in which to implement a tool which can support the distribution and revision of documents, programs, etc., and do so without any special actions on the part of the producers.

3. Advice on Building Heterogeneous Distributed Operating Systems -- by Mark Marcus

3.1 Introduction

3.1.1 Motivation

This article offers some advice to the would-be designer and builder of a heterogeneous distributed operating system (hereafter referred to as H-DOS). An H-DOS runs on a network of dissimiliar computer systems. The H-DOS user has an easy and integrated access vehicle to the resources of each computer system attached to the H-DOS. A computer system is defined in this article as being composed of the hardware, the software, and the native-operating-system.

A large amount of the information presented here has been obtained from the National Software Works Project [NSW 83] (hereafter referred to as NSW). NSW was an ambitious prototype for an H-DOS which ran on the ARPANET from 1976 through 1983. In 1983, NSW had connected the following ARPANET computer systems: an IBM 360/370, Digital's TOPS-20, Honeywell's MULTICS, and a UNIX running on a PDP-11/70.

3.1.2 The Present Situation

There exist many different kinds of computer hardware -- mini- and micro-computers, mainframes, number crunchers, high resolution graphics workstations, personal computers, etc. -- each one fulfilling a specific real-world need. For example, a mainframe may be appropriate for a system that keeps track of all foreign military vessels in the Atlantic Ocean, because of the large amount of data involved in such a job; however, a micro-computer may be more appropriate for an on-board system that navigates a missile. And just as there is a need for different types of hardware, so is there a need for different types of operating systems. Some operating systems may be made highly efficient by not providing many operating system calls; while other operating systems may be less efficient but provide excellent support for software development. Some operating

systems may be best suited for the master file update algorithm (e.g. financial packages) while others may be more suitable for process control (e.g. robots for manufacturing).

One drawback in having all these different computer systems around is that they require that a user learn the idiosyncrasies of each environment. At present, different computers have different operating systems, which in turn have a certain class of software that runs best on each of them, with the unhappy result that the user cannot sit down at a single terminal and orchestrate the use of different software in an integrated way.

For instance, a mainframe may be more suited for running an Ada compiler with the object code intended to be downloaded to a mini or micro for execution or debugging. Or a superior editor may be available on one computer, a superior compiler on another. The first step towards accommodating such factors as the above has already been completed as demonstrated by the existence of computer networks such as Arpanet, Tymnet, Csnet, and Ethernet.

Network software is designed in several layers; each layer providing a service to the layer below it. A popular layering model is the ISO/OSI reference model for network architecture. The first layer simply provides an electrical connection. The next layer makes sure that bits get from point A to point B in a reliable fashion. In a number of network architectures, the network layers are advanced enough to provide two services which we will call FTP and Telnet. FTP allows for file transfer among computers of the network, and Telnet allows for a user to attach and log an interactive terminal into any computer on the network as though he were a local user. These services have some limitations in that file structure, and assumed terminal characteristics differ from one system to another. Many problems are solved by establishing a connection between tools (e.g. compilers, editors, etc.) and system resources via virtual terminal and file structuring. Before crossing computer boundaries, the actual structure is mapped onto a virtual structure. However, services like FTP and Telnet do not provide easily shared resources. For one thing, file transfer is tedious and the different idiosyncrasies of each system must still be mastered. There is no provision for a uniform file system accessible across all computers in a network as if each file were local to the computer.

As mentioned above, NSW was a prototype system that provided uniform access to a number of widely differing computers -- allowing a person to run programs on several different computers, and to have files produced by one tool be used by another tool on a different system with the file transfers being transparent.

Many engineering lessons were learned in the NSW project. Below is a list of some of the items to think about when designing and building an H-DOS:

- Failure Modes and Uniform Error Handling
- Virtual Firm Connections
- Procedure Call Protocol
- Distributed Data Bases
- H-DOS/Tool Interface
- Accounting and Logging
- Demand Paging
- Command Scripts
- Response Time
- File Storage
- Immersion (i.e., developers in system)

The ideas contained under these topic headings often show up or blur across topic boundaries. Let the implementor beware.

3.2 Failure Modes and Uniform Error Handling

Say there are 6 computers on a network. Then the number of states this network can be in with each of its computers in

either the up or down mode is 2^6 (i.e. 64). In general, this number is 2^K, where K is the number of computers on the network. In other words, if you would have 16 computers on the network, you would have 65,536 possible states of the network. Moreover, there may be many failure modes -e.g. DOWN, TOO-SLOW, MALICIOUS, BLACKHOLE, etc. So, the number of states the network can be in is really $(M+1)^K$, where M is the number of failure modes and K is the number of computers in the network. This is much larger than the 2^K above. Software running on this network would have to take these failure modes into account; if not, big problems could arise.

What is the symptom of software that doesn't properly take into account failure modes? Most likely, such software would either hang, crash, or deadlock when it first encountered a failure.

Take the following scenario: a computer is about to be pulled out of the network -- no one will be able to communicate with it. Before this happens, an unsuspecting user of the network (i.e. a computer process or a human being) makes a request of the system that requires communication with this soon-to-be-incommunicado computer. Suppose this request could take between one and five minutes to complete. What happens when this computer goes "down for the count" (i.e. is pulled out of the network) after this request has been issued. One solution would be to place a 5-minute timeout in the caller computer. The caller computer decides to wait the maximum amount of time for a transaction before taking any action. This solution, while it will eventually detect the problem, does so in a worst-case fashion (consider a request that has a maximum transaction time of 60 minutes, but most of the time takes 20 seconds).

Either the user is standing next to the "fallen" computer and sees that it is down and wonders why his service-initiating computer can't detect that fact; or the computer is out of sight and the user looks helplessly at his terminal wondering what is going on (especially when the user learns that he could be waiting on a failed component or may be waiting for a legitimately long transaction time).

See the "Virtual Firm Connection" section for a quick method for detecting when computers become inaccessible over the network. The reason for inaccessibility could range from the computer crashing to the network communication lines being broken.

Another symptom of improperly handling failure modes is inconsistent and improper messages being sent to the user's terminal. In NSW, for example, if a user had requested the use of an editor on a crashed computer, he would sometimes be shown a sequence of messages indicating the successful start of the editor followed by a timeout message indicating the unavailablility of the editor; instead, he should initially be notitifed that the resource is unavailable. If the specific reason for failure is known, and this information is useful to the user, it should also be mentioned on the user's terminal.

Failure modes are only a subset of a larger concern about error handling in general. It is important that a scheme to handle failure modes and errors is designed into the H-DOS from the start.

Before delving into error handling, I would like to introduce the concept of rapid prototyping. A rapidly prototyped version of a product is usually considered to be a version of the product that is built quickly, minus the bells and whistles, with the rough edges still in place.

Rapid prototyping is just as applicable to the building and design of an H-DOS as to any other product. In a sense, handling failure modes and rapid prototyping go hand in hand. When designing any system, if a part of it is working quickly, potential customers can then see how the design fits their needs. One way of rapid protoyping is not to handle all of the failure and error modes explicitly. This method is akin to Ada's exception handling facility. The basic idea is that, during the first draft of the code, all errors are just "passed up" all the way to the user. If a routine encounters an error in this first version of the code, it simply returns an appropriate error message to its calling procedure. That procedure in turn returns the error message, along with a note that the error has passed through this procedure. In the end one will have a trace of

procedures ending with the one that caused the problem. How much of this calling chain is passed to the user is implementation dependent.

This scheme allows for quick production of code, because the normally large percentage of code treating errors does not have to be produced right away. However, the structure is there to be filled in with the proper error handling code when a more finished product is developed.

Here is an example. At first there is no error handling code. Now, suppose you have a disk error. In this initial, unfilled version of code, the user is ultimately told that the transaction requesting a disk read failed. How this information is presented to the user is an arbitrary design choice. Later, as code is filled in to handle specific errors, a different scenerio might result. For instance, the same request was made with the same disk error; however, now an error handling routine decides to get a duplicate copy of the file from some other computer; the user may not even be notified. The error will be recorded in a log to be read by system operators.

In filling in these error handlers, it should be noted that it is important to assign responsibility for error handling to code authorized to make the decision. And, as in business management, the code performing decision-making should be at the lowest level possible. For instance, a checksum problem should be handled by the link level protocol, not by an application program. And conversely, the decision to search for another copy of an inaccessable file should not be handled by the link protocol program.

It is also appropriate for higher levels to be informed of "failure trends" by lower levels, e.g., to emit "Please wait" messages.

3.3 Virtual Firm Connections

Many computer networks are packet-switching networks. A

common way for two processes on different computers on a packet-switching network to talk to each other is by establishing a virtual connection. When a process on computer A wants to talk to a process on computer B, it asks its local network software for a virtual connection. When this virtual connection is established, the process on computer A can simply send the data down the software connection without having to specify the host and process name everytime. From a software point of view, there is a connection between processes on different computers when, in fact, there is no dedicated physical connection on packet-switching networks.

The detection of a break in virtual connections in most protocols only takes place when there is an attempt to send a message over the connection (e.g. as in the TCP protocol). This ultimately translates into a sluggish response time of the H-DOS to the user.

Suppose a process control system is built on top of an H-DOS. And suppose a user of the network decides to fill a tank with propane only if there exits a connection to the computer controlling the fire-extinguishing hardware. If a conventional virtual connection were established with the fire-extinguishing hardware, there would be no detection of a break in communication with the fire extinguisher until there was an attempt at using the connection. Finding out you can't control the fire extinguisher when you need it could be fatal.

In general, when the ability to communicate with another process is lost, the network will perform more smoothly if it can respond to the break immediately. Of course this is not always the case; but in those cases where it is desired, <u>virtual firm connections</u> should be used.

A virtual firm connection (a term coined for this article) is a virtual connection that can detect breaks when they happen. In packet-switching networks there is only one way to do this, and that is to periodically test the connection for continuity with the transmission of dummy messages. These dummy messages are sent by the layer of software below the application layer. The application software does not get involved with the transmission of these dummy messages. Only when a connection break is

detected is the application software notified. This technique is used quite often in critical process control areas.

During the construction of an H-DOS, it is often helpful to use the virtual firm connection mechanism. Nevertheless, the decision to use virtual firm connections over virtual connections must be carefully thought out. There is an overhead associated with the use of virtual firm connections, the cost of sending periodic dummy messages. This cost must be weighed against the need for quick detetection of communication breaks given by virtual firm connections. In some cases it might be found that the overhead may not be worth it, especially if the reliability of the connections is statistically very high.

3.4 Procedure Call Protocol

A typical method of getting two processes on different computers to talk to each other is by sending a block of data (i.e., datagrams) between them. The receiving process must parse the datagram and hope the bits really represent the desired data types.

Another approach, separately written about by James White [White 76] for the Xerox System Integration Standard and Stephen Schuman [Schuman 81] for the Ada environment, is to communicate with foreign processes by a procedure call, just as you would for a local procedure call. This method provides for parameter checking and all the other benefits associated with the procedure calling mechanism. Of course, from a procedure call one must allow for error returns that indicate a communication failure. A pleasing quality of this is that the error can be handled or not handled by the same method outlined in the "Failure Modes/Uniform Error Handling" section.

3.5 Distributed Data Bases

It is well worth spending time on constructing or obtaining a good distributed database system. Issues involving construction

of distributed database systems can be found in the literature (for instance, [Ullman 82]). In the H-DOS environment, a distributed database system can be used to full advantage in several areas:

- as part of a file system
- for storage of H-DOS's users profiles
- for logging and accounting data storage
- as a tool for application programs

Many of the kinds of data normally found in a conventional operating system (e.g. files, user profiles, etc.) are scattered about the numerous computer systems of an H-DOS. A distributed data base system glues this data together and to a large extent removes concerns over computer crashes and communication failures. In other words, distributing data causes problems, so why not create one piece of software that deals with those issues in a uniform way and be done with it. In NSW this was not the case; each subsystem had its own way of dealing with the integrity of its distributed data, with some components not doing as good a job as others.

3.6 H-DOS/Tool Interface

There are two ways to go about building an H-DOS:

1. Have a guest level implementation which implements an H-DOS on top of the already existing operating systems (i.e. MVS, TOPS20, Unix, etc.)
2. Have H-DOS supplant the local OS.

Having a family of operating systems specifically constructed for H-DOS has the nice property of being more efficient than (1) the so-called guest level implementation. The advantage of (1) is that it is less costly to build an H-DOS out of already

existing operating systems, and it allows a user familiar with a local operating system to use the H-DOS only as an access route to services not found on the local host. The user can still use familiar tools for problems that require services provided by the local system, while using H-DOS for services provided non-locally. This section will only discuss (1), the guest level implementation scheme.

Ideally, a tool such as an editor should be able to be accessed from both the local operating system environment (e.g. UNIX) and the H-DOS environment (e.g. NSW) with equal ease. If H-DOS and the tools are developed with this in mind, this is no great technical feat to accomplish. However, to retrofit existing tools into H-DOS without modifying the tools I/O is a difficult problem.

Two techniques have been tried: a backend (BE) and frontend (FE) approach. Both have had only moderate success. The problem in both the FE and BE techniques centers around redirecting a tool's request for an operating system service from a local operating system call to an H-DOS call.

In the FE approach, the H-DOS could monitor the stream of characters entering and leaving the tool. When H-DOS "thinks" a filename was being requested, it could replace it with an H-DOS file mapped into a local file. In the BE approach, if the local computer could trap the local operating system calls, all local system calls handling files could be redirected to the H-DOS (as is done in NSW). But how should temporary file requests be detected? And what if the tool itself does file name parsing, with local filenaming syntax?

Both FE and BE approaches require sophisticated techniques to properly determine how and when the H-DOS should stick its nose into the tools activity. Also, the mapping of an H-DOS filename into a local filename space is non-trivial, as the syntax of both type of filenames can be quite different.

The solution -- if you use existing tools -- is to recode the I/O to connect to H-DOS. A library of routines could be made up to make this process less painful. In most instances, tools

request services from their operating system by way of calls with names such as OPEN, CLOSE, READ, WRITE, SEEK, etc. Tools could be recoded to call H-OPEN, H-CLOSE, H-READ,H-WRITE, H-SEEK instead. These H-XXXXX calls would detect, possibly at run-time, that they were running in either the H-DOS environment or the local operating system environment, and then they would service the requests accordingly.

3.7 Accounting and Logging

During the running of H-DOS, two kinds of information must be gathered: (1) usage charges to collect for accounting purposes and (2) diagnostic data for maintenance and operation of the H-DOS system. Because H-DOS is distributed, locations where the information is being gathered are also distributed. Information gathered during H-DOS runtime can be stored in a distributed database system, taking advantage of the database's reliability. This also has the advantage of being able to use the power of the database's query language. Questions can then be asked such as: Who was the last user of a tool before H-DOS crashed? Also, statistical questions could be asked, such as: What percentage of users use the Ada compiler?

Although gathering information for accounting is not difficult, finding a reasonable charging scheme is. The problem stems from the fact that an H-DOS may store or choose data paths that are optimal to it by its own criteria. Is the user to be charged for storing multiple copies of a file when H-DOS decides to do this for its own purposes? Does one break out the charges for using the communication lines, the local computer, the computer whose service is being used, etc.

One lesson learned from NSW is to think about accounting during the design phase. It is hard to retrofit such a feature in an H-DOS.

3.8 Demand Paging

In an H-DOS, where text files may be in different forms (i.e.

ASCII, EBCDIC, etc.), file transfer between systems usually involves changing the structure of the file. As hinted at before, this is done by using a virtual text file structure. The original file is mapped into a canonical structure and then mapped to its destination structure. This method avoids the (N) multiplied by (N-1) number of translation schemes needed if, for each pair of computers, H-DOS directly translated a file into the structure of each destination computer.

This approach, however, led to not making good use of demand paging in NSW. Demand paging means only load into memory those pages (i.e. sections of a file) which a tool needs at that moment. If the tool needs other parts of the file later, it can request them later. In NSW, when a request to use a non-local file was made, the translation scheme was invoked on the whole file. The tool did not get access to any of the file until the whole file was translated. This slows up the tool unnecessarily. For instance, an Ada compiler could not start its processing on a file until the whole file was translated. The Ada compiler itself may be slow. In fact, if the rate of translating and transporting the file and the rate of the Ada compiler were equal (an unlikely event) and the Ada compiler could start compilation after receiving the first page, the compilation for a long file will be almost complete by the time the whole file gets translated and transported.

In short, if the tool is on one computer and the file to be used with the tool on another -- do not wait for the whole file to transfer before starting the tool. Demand Page.

3.9 Command Scripts

In an H-DOS environment, command scripts are essential. They are, as in any operating system, valuable time savers which are useful in regression testing.

In NSW, one difficulty encountered in the command script implementation involved the command script characters getting lost when the command script called for changing from one tool to another. This points a general engineering issue: in a distributed system, care must be taken to make sure that the data source "knows" when the data sink is prepared to accept data.

3.10 Response Time

One complaint about NSW was that at times it was too slow. This occurred because the resources the H-DOS was integrating were heavily loaded. The varying amount of access time to a user is disconcerting. The solution to this is either to have a system with enough capacity to support an H-DOS, or to have a mechanism that can warn users of the response of certain requests based on the load of the computers needed to complete the request on H-DOS.

3.11 File Storage

In the guest level H-DOS system, where are the files stored? Are they stored in local user directories and somehow pointed to by the H-DOS system? Or, is there a separate file system for H-DOS?

If H-DOS points to files in a user's local directory, the user has the benefit of being able to create a file with his local operating system and then to immediately access this file in the H-DOS envirionment. However, there is a drawback in this method. The H-DOS may want to move the storage of files to certain computers based on its own needs (i.e. space needs, file is frequently used at a particular computer, etc.). A compromise should be implemented. The H-DOS should have its own file system, but there should be a command to map an entire local directory into H-DOS (at which time the local user will no longer be able to modify the directory contents locally). A reverse process of unmapping should also be provided.

3.12 Immersion

Immersion is having the developers use, as soon as possible, the system which they are building. This is more of a statement

of good practice than anything else. If the developers have to use a system, it is more likely to become a comfortable system. NSW did not employ immersion.

3.13 Conclusion

As the above list of issues suggest, building an H-DOS is not a matter of creating new technology but of engineering existing technology. NSW did, in fact, prove the feasibility of blending these technologies.

NSW, as a prototype, detailed many of the pieces that need to be carefully designed in the next generation of H-DOS systems. These include the topics discussed in this article. Some of the technologies, such as distributed data base technology, needed to build a successful H-DOS have matured since the start of the NSW project.

Many of the issues stated in this article are dealt with in other distributed systems. It is the purpose of an H-DOS to solve these problems and let application programs and users fully utilize distribution without engaging in these issues.

The need to interact with different computer systems is ever increasing, in both the military and commercial sector. The concept of an H-DOS will play a significant role in supporting this much needed interaction among heterogeneous computer systems.

4. The NSW Object-Naming Technique -- by Kirk Sattley

4.1 INTRODUCTION

One of the little-publicized aspects of the National Software Works (NSW) effort was the development of a naming scheme for catalogued objects in an environment where the number of named objects was expected to be very large. As the NSW final report is being written, it behooves us to put on the public record a sympathetic description of this technique in the belief that the reader may one day encounter a situation where these ideas will be of use.

The striking feature of the system that ultimately evolved is that it is neither hierarchical nor relational, in the current senses of those terms when applied to file systems or databases:

- The naming scheme can be used in a hierarchical fashion, in that the total system namespace can be divided into regions, which can themselves be subdivided, and so on; but it is in fact a "flat" system: it contains no "structural" objects as do hierarchical directory systems -- the names are not "path names".

- The scheme can be used in a relational fashion, in that sets of objects can be selected according to "orthogonal" properties of their names (without regard to any hierarchical interpretation the names might also have), although there are no named "relations" or named "attributes" (in the naming mechanism proper) -- the names are not rows in fixed-width tables.

- In addition, this scheme can be used as a keyword-indexed retrieval system, retrieving objects that bear particular tags, independently of any hierarchical or relational place-value the tags might have.

The system presented in this paper is the "closure" of the resource-catalogue scheme as it was actually running at the end of the NSW effort. The designing and planning went on

continually throughout the development of NSW, and the last released system contained several new features, plus mechanisms anticipating the next level of enhancement. I shall describe here the system we were aiming toward as if it were complete, and at the end of the paper add a short apologetic section delineating the described features which were never actually released.

The present paper attempts to present the principal features of the naming scheme in an order which minimizes forward references. Each section describes an aspect of the system and discusses its use and its interaction with previously-described features. The first subjects discussed are the distinctions and relations between names, catalogue entries and objects of the system, the form of those names, and the form of designators for sets of names -- sets of names being the conceptual centerpiece of the entire scheme. Then the use of these set-designators for describing access permissions is discussed, along with the facilities this provides for designing a namespace for a group of users; this discussion then leads into illustrations of the "relational" usage of the naming scheme, mentioned above. This provides material for exploring further implications of overlapping subspaces as defined by permissions, and for illustrating the keyword-retrieval uses of the naming scheme. The use of non-name information in the object catalogue for additional selectivity in denoting sets of objects is mentioned (for completeness, as it is distinctly a secondary mechanism). The last major structural feature of the system is then introduced, one which allows users to define and name arbitrary subspaces as name-interpretation contexts, but which leads to a nice disquisition on the desirability of imposing a restriction solely for the purpose of simplifying the user's mental image of the system. A notation is introduced to further shorten names by allowing the user to denote his favorite default name-resolution context by not mentioning its name. A logically unnecessary but useful feature of the system is next described, by which each user can have his own, private, space of short names -- analogous to a personal directory in other systems. One further notation is introduced, which permits the user to signal easily that a designator he uses is to denote a single object rather than a set. After a brief exploration of the use of plural designators for repeated source/target operations, the paper concludes with the promised description of the actual status of the system as embodied in the last-released version of the National Software Works.

The intended reader of this paper is anyone with an interest in shared-namespace systems of broad compass, particularly someone who has encountered the problems and issues involved in devising naming conventions for a many-person project. Some of the sections are concluded by an "Implementation Note"; these are rather more terse and technical, intended to reassure the more skeptical reader as to the implementability of the system.

The prescriptions given here for the use of particular special characters and notational conventions are arbitrary, of course: what is essential is the concepts that are represented, not the puctuation marks which encode them. Examples are shown in uppercase letters for visual distinctiveness in the text, not as a matter of general preference.

4.2 OBJECTS, CATALOGUE ENTRIES, NAMES

In the total operating environment of a large multi-user system, there will be many entities which need to be individually designated -- which need to have names. A large, central, class of these entities (the ones the system is "about", in some sense) will have names constructed according to some scheme or principle of name-building, which embodies assumptions about the uses of names, the accessibility of the named entities, and relationships between the entities. The purpose of this paper is to describe one such scheme, approximately as it has been used in the National Software Works ("NSW", hereafter) project.

We use the term '<u>object</u>' for the class of entities to which the naming scheme applies, to distinguish them from other name-bearing entities in the environment. In NSW, the objects were originally all files, and the naming scheme was thought of as a "file system"; later it was extended to include tool programs and devices, collectively called "resources". Some of the non-"resource" entities in the system were user identifications and network hosts.

As far as the system is concerned, an object exists only if there is a <u>catalogue entry</u> for it in the central resource catalogue. Just as in most file systems, when the name of an object is submitted for the purpose of gaining access to the object, the interpretation of that name by the system leads to a

data record which is then passed to some different mechanism for performing the actual access. In the NSW, this decoupling is more complete than is usual in single-host operating systems, since the catalogue information is used to direct an agent process on some other host (in the general case) to deal with the actual object.

Hence, from the programming point of view, the name of an object is really the name of its catalogue entry, in the direct sense that what is returned from a name-lookup request is (a pointer to) the catalogue entry found with that name, or the indication that no such catalogue entry exists.

The present naming scheme is built around the ability to designate sets of names with a single expression, which is called a spec. Extending the principle that the lookup of a name returns the catalogue entry bearing that name (if it exists), the lookup of a spec returns a set (perhaps empty) of catalogue entries that bear names belonging to the set.

A spec, then, serves primarily as a designator for a set of names: given a name and a spec, we can determine whether the name belongs to the set described by the spec just by comparing the two as strings. Secondarily, the spec serves to designate the set of all those catalogue entries which carry names belonging to the set, as determined by the lookup procedure. And finally, after another step, the spec can be said to designate the set of objects whose catalogue entries were returned by the lookup. Note that the membership of this set of objects will change from moment to moment as objects are added to and removed from the system, but the abstract set of names does not change: a name belongs to the set, or not, independently of whether the system actually contains any object carrying that name.

A formally correct name can exist before there is a catalogue entry for it -- as when the user gives a name for an object he is about to create. Similarly, a catalogue entry may exist before there is any physical object for it to refer to -- as when the name to use for an output file is given to a batch program before it starts.

We shall use the term 'namestring' when it is useful to emphasize the representation of a name, abstracted from any entities it might denote.

4.3 THE FORM OF NAMES

A namestring in this system consists of a sequence of identifiers separated by periods. Since the system contains a great many objects that need names, namestrings tend to be long; but there are several mechanisms which allow users to employ short names almost all the time.

The fact that a name is a sequence of identifiers means that order is significant: the namestrings 'A.B.C' and 'A.C.B' denote different objects.

There are no implicit objects: The fact that an object with the name 'A.B.C' exists does not imply the existence of any object named 'A.B'; and if an object named 'A.B' does exist, it is just another object in the catalogue -- NOT a "directory" which "contains" an object named 'C'.

Further, the fact that two namestrings are similar -- such as 'A.B.C' and 'A.B' of the preceding paragraphs -- does not imply the existence of any system-maintained relationship between the objects they denote; it might imply that the user (or process) that gave the newest one its name considers them to be related in some way, and has chosen to record that relationship in the choice of the name.

The identifiers making up a namestring are called name components.

The namespace of the system is the set of all namestrings, subject to whatever pragmatic limits might be imposed on the length of identifiers and the number of identifiers in a name.

Implementation Note:

The catalogue is implemented in a quasi-associative memory scheme -- actually, of course, an inverted-list information retrieval system. When an object is entered into the system, a catalogue entry for it is created, and a pointer to that catalogue entry is added to the inversion-lists for each of the components in the object's name. The order of appearance of the name components has no effect on this process, so, for instance, a request to retrieve the object named 'A.B.C' will at first produce three lists of entry pointers, one for each of the separate name components, listing those entries whose names contain that component. These three lists are then intersected, and the resulting list contains all entries bearing all three name components, but in any order -- A.B.C, A.B.C.D, A.C.B, B.A.A.C, and even C.B.A.D.E.

Finally, this list of candidate entries must be culled to select only the one actually requested -- that is, the one that has all name components in the right order and begins and ends with the correct components. Each catalogue entry pointed to on the intersection-list is fetched, and the namestring found there is compared to the name requested, until an exact match is found.

The fact that a final culling always needs to be done leads to an interesting aspect of this inverted-list system: The list of entry-pointers for a particular name component is located by hashing the text of the identifier, as one might expect; however, in this system, there is no need to worry about "collisions" (two distinct identifiers which happen to hash to the same value) because any "false drops" which an un-disambiguated collision might generate will automatically be weeded out by the final culling operation. So it is only necessary to keep one pointer-list for each possible value of the hashing function, and not one for each identifier which appears as a name component.

4.4 SETS OF NAMES, SPECS

As we mentioned before, the central theme of this naming scheme is to provide simple means for designating sets of names, and hence, of objects. Even for simple operations, it is approximately true that one designates a single object by describing a set which happens to contain only the object desired. Most of the time, the individual programmer needs to name only single objects and may not think in terms of sets of

objects, much less sets of names. Nevertheless, even a programmer who always works on one subroutine at a time needs such operations as reviewing the names of available objects ("listing the directory"), "connecting" to a new working context, checking for the presence of new objects of interest in an area shared with other workers; all of these can be cast in terms of operations on sets.

This facility for designating sets of objects is used not only for repeated performance of an operation on a number of objects, but also, and more importantly, for assigning access rights and for defining working contexts. Anyone concerned with project management or coordination will find frequent use for these and other operations that take sets of names as operands.

The sets of names which can easily be described in the system are, unsurprisingly, those which are similar namestrings -- similar, in that they have parts in common.

A designator for a set of names consists of that part of the namestrings which all members of the set must have in common, decorated with "wild card" notations for the variable parts. The wild-card symbol is the asterisk, '*', and is to be thought of as standing for any number of name components, including none at all. As mentioned above, a designator for a set of names is called a 'spec'.

A manager or programmer, "designing" a portion of the namespace, will choose its naming conventions in such a way that it is easy to write specs that denote the sets of objects that will be mentioned most frequently. This is what is done in hierarchical systems, of course, except that there, the similarity relation between namestrings is "has an initial sequence of name components in common". In this system, the relation is less restricted, but harder to express in a phrase. Let's consider some examples:

- The spec 'PROJ3.*' denotes the set of names that begin with the component 'PROJ3'.
- The spec '*.SOURCE' denotes the set of names that end with the component 'SOURCE'.

- 'PROJ3.SOFTWARE.*.A_SPEC' denotes the set of names that begin with the components 'PROJ3' and 'SOFTWARE' (consecutively in that order) and end with the component 'A_SPEC'.

- 'PROJ3.*.DOC.*' denotes the set of names that begin with the component 'PROJ3' and contain the identifier 'DOC' as a name component somewhere in the rest of their names.

- The spec '*.PUBLIC.*' denotes the set of names that contain the component 'PUBLIC'.

- '*.PUBLIC.NOTICE.*' denotes the set of names containing somewhere the two components 'PUBLIC' and 'NOTICE', consecutively and in that order.

- '*.DOC.*.PUBLIC.*' denotes the set of names that contain the components 'DOC' and 'PUBLIC' anywhere, but in that order -- at least one 'DOC' must appear to the left of some 'PUBLIC' in each namestring in the set.

As these examples indicate, the "part" of the namestrings which appears in the spec for a set need not be a consecutive stretch of name components, but just a subsequence -- a subset of the name components, with relative ordering preserved. The spec serves as a pattern for the namestrings in the set: any namestring which can be produced by "filling in the blanks" in the pattern-spec is a member of the set.

To put this "pattern" idea more formally: A namestring belongs to a given set if it corresponds to the spec for the set under an order-preserving mapping which matches every name component in the spec with an identical one in the namestring, and which maps '*'s in the spec into sequences of zero or more consecutive components in the namestring. When this correspondence does hold, we also say that the spec covers the name.

This correspondence of the '*' with an arbitrary sequence of name components means that it is meaningless to have two consecutive '*'s in a spec; hence it is considered an error, in order to remind the user that he has misunderstood something. At

various times, consideration was given to additional "wild-card" symbols to represent space for at least one, or for exactly one, name component. These were felt to add more complication than usefulness, though one could imagine namespace structurings for which they would be valuable.

Implementation Note:

The inverted-list retrieval scheme applies quite straightforwardly to the retrieval of sets of objects: the lists of catalogue-entry pointers for all of the name components given explicitly in the spec are intersected, and each candidate in the resulting list is checked for conformance with the spec under the correspondence rule given above.

This checking always reduces to testing whether a given spec covers a given full namestring, and can be done by a rapid string operation: the most compact form is a simple recursive routine, where the depth of recursion is equal to the number of non-terminal '*'s in the spec.

4.5 SPECS AS A VEHICLE FOR RECORDING PERMISSIONS

Thus we have a view of the system namespace as the set of all permissible namestrings, in which an enormous number of overlapping subspaces can be defined by means of simple string-similarity relations. One of the principal motivations for this approach is to be able to use these subspaces as the domains of application of access permissions.

A permission consists of two pieces of information: a spec designating a subspace of the system namespace, and a list of one or more rights which the holder of the permission may exercise within that subspace. The rights are of the traditional sort:

See, Enter, Delete -- a catalogue entry;
Read, Write, Execute -- the content of an object.

When a user is logged in, the system has available the set of

permissions which that user holds. (The question of how the user acquired the permissions is an interesting topic for discussion, but -- as handled in this system -- separate from the subject of the present paper. See [Warshall 80].)

Whenever the user requests an operation involving a set of names, the system, in effect, retrieves all the catalogue entries in its database whose names lie in the designated subspace, then tests each one to see if it is covered by some permission the user holds. (In formalistic language, then, the set of entries retrieved under a given spec for a given user is the intersection of the set corresponding to the request spec with the union of the sets described by the spec portions of each of the user's permissions.)

Implementation Note:

The process of running a permission-check on the object name in each catalogue entry can be speeded up by preprocessing the permissions in the user's login-session record so that no permission will be checked to verify a particular type of access if the user also holds another permission with the same access right and a more inclusive domain. Thus, in the example given in the next section, the permission with domain PROJ3.SENSOR.*.SOFTWARE.* would be used for checking only for accesses requiring Enter, Delete, Write rights, since the non-destructive See, Read, Execute rights are provided by other permissions with larger domains.

4.6 NAMESPACE ORGANIZATION -- HIERARCHICAL USAGE

It is this use of namespace specs as permission domains which permits -- and encourages -- a hierarchical partitioning of the total namespace: Each major group of users ("project") receives full rights to one or more subspaces determined by specs with a single, mnemonic, first name component: 'PROJ3.*'. Members of other projects might hold See and Read rights within this namespace, but not Enter, Delete, or Write.

The manager of Project 3 would then divide the project's namespace into subareas, corresponding (for instance) to the

structure of the project itself: 'PROJ3.SENSOR.*', 'PROJ3.ACTUATOR.*', 'PROJ3.HOUSING.*'. Members of the project would receive permissions giving them the appropriate rights in each of these subspaces.

Thus the left-hand end of the names of the objects in a system tends to look like a simple hierarchical scheme, with all its advantages of conceptual straightforwardness, and its difficulties of determining the best order to use for subdividing the categories named by the components. In our example, should the project's namespace first be divided into major subassemblies of the final product, as suggested above, or should its principal division be according to the human activities involved -- e.g., 'ADMINISTRATION', 'ENGINEERING', 'PROGRAMMING', 'TECH_WRITING'? This problem arises in one form or another whenever a strictly hierarchical breakdown must be used. (The AIE KAPSE design [AIEK 82] has neatly avoided this problem by allowing the nodes at each level of its hierarchy to be multi-dimensional.)

Such problems are mitigated in this system, since it is entirely possible to construct orthogonal subspaces. A management plan in which the subprojects were organized around the major subassemblies of interest (sensor, actuator, housing) would organize its namespace with fixed second name components as suggested above, then also give permissions to subspaces within PROJ3.* that cut across the subproject partitioning. For instance, a programmer working on the sensor software might have the following permissions:

PROJ3.*.DOCUMENTATION.*	(See, Read)
PROJ3.SENSOR.*	(See, Read)
PROJ3.*.SOFTWARE.*	(See, Read, Execute)
PROJ3.SENSOR.*.SOFTWARE.*	(All rights)

What does the phrase "organizing a namespace" mean in this environment? There is no way for the manager of Project 3 to tell the system directly "The only allowable name components following an initial 'PROJ3' are 'SENSOR', 'ACTUATOR', and 'HOUSING'." A namespace is "organized" by the permissions which are in existence: initially, the manager of Project 3 is the only user in the system with (Enter, Write, Delete) rights in the space PROJ3.*. As project manager, he will be entitled to give

sub-permissions to members of his project. He may give out any number of copies of (Enter, Write, Delete) permissions in the spaces PROJ3.SENSOR.*, PROJ3.ACTUATOR.*, and PROJ3.HOUSING.*, but as long as he refrains from giving out any (Enter, Write, Delete) permissions within the PROJ3.* namespace with a different second name component, and doesn't use his broad permissions to create any catalogue entries in other subspaces, the namespace is effectively "organized" as intended.

4.7 QUASI-RELATIONAL USAGE

The permissions 'PROJ3.*.DOCUMENTATION.*' and 'PROJ3.*.SOFTWARE.*' in the previous section hint at the "quasi-relational" use of the present naming scheme, which was mentioned in the Introduction. In line with our previous examples of Project 3's namespace, let's say that the manager intends his object names to have the general form

PROJ3.<subassembly>.<expertise>.<component>.<details>

where <subassembly> may be 'SENSOR', 'ACTUATOR', or 'HOUSING', <expertise> may be 'HARDWARE', 'SOFTWARE', or 'TESTING', <component> is the proper name of some separable hardware component or software module, and <details> is any additional information needed to uniquely specify the object (file, probably).

This view of the project's namespace is clearly relational in spirit -- the suggestive nouns in angle-brackets are the names of the "attributes" in the PROJ3 "relation", and the possible values of these "attributes" have been listed or described. The manager will be able at will to view the state of the project along any of several dimensions, as suggested by these examples:

```
PROJ3.*.HARDWARE.POWER_SUPPLY.*
PROJ3.SENSOR.*.POSITIONING.*
PROJ3.*.HARDWARE.*.SPEC_SHEET.*
PROJ3.*.SOFTWARE.*.OBJ
PROJ3.HOUSING.TESTING.*
```

The system itself, for better or worse, provides no mechanism for enforcing these quasi-relational conventions: there is no way of verifying the value to be placed at a given position in the name of a new object (and there is no concept of position except as defined by adjacency and order with respect to anchored elements of a spec).

But this is not without its advantages. Imagine the project a month after its beginning, when the manager and his chief engineers decide that the Actuator subassembly should "really" be thought of as two separate work areas, Electrical and Hydraulic, as far as Hardware engineering is concerned, and for some, but not all, of the Software. In the present naming scheme, there is no danger in splicing the subspaces PROJ3.ACTUATOR.ELEC.* and PROJ3.ACTUATOR.HYDR.* into the naming conventions for the project, without modifying any of the existing operating rules or changing any specs which might have been written into management report generators. The fact that, for instance, 'HARDWARE' will sometimes be the third, and sometimes the fourth, component in an object name doesn't keep it from playing its role as a value of the <expertise> attribute in the naming scheme.

4.8 OVERLAPPING SPACES, ANCHORED PERMISSIONS

All the permissions discussed so far have had "anchored" domains -- that is, their domain specs have started with one or more explicit identifiers, not with asterisks. What about "unanchored" permissions? For a specific example: We could give everyone in the system a (See, Read, Execute) permission for the domain *.PUBLIC.*; this would allow anyone to "publish" a document or program merely by including 'PUBLIC' as a component in the name of an object that they are Entering into the catalogue.

Note the working of "orthogonal" permissions: a user has the right to perform any access permitted by any of his permissions. I don't need Enter rights to *.PUBLIC.* to use the identifier 'PUBLIC' in a name I create: if I have Enter rights to PROJ3.SENSOR.*.SOFTWARE.*, I am entitled to create an object in the subspace PROJ3.SENSOR.* as long as its name also contains the component 'SOFTWARE', and hence I can name an object 'PROJ3.SENSOR.REDUCTION.SOFTWARE.PUBLIC.EDGE_FINDER.SOURCE'. And conversely, no matter how private Project 3 considers its PROJ3.*

space, if outsiders hold a *.PUBLIC.* permission, they can See this object, and thence infer the existence of Project 3.

To allow other than very restricted administrative use of unanchored permissions, then, would be dangerous except in very open systems. The users would have to learn, and remember, that the privacy of their project and OWN spaces could be breached by the use of any of a certain small set of universal words: if they used any of those words in the name of an object, then that object would be accessible to users who did not hold any "proper" permissions within the space.

Nonetheless, even if unanchored permissions are disallowed in the entire namespace, the use of permissions "floating" relative to a sub-namespace can be quite advantageous. Project 3's chief technical writer might well hold an All-rights permission with domain 'PROJ3.*.DOCUMENTATION.*', which conveys rights cutting across all sections of the project.

Note also that no new mechanism is required to disallow permissions starting with an asterisk: such a permission could be given only by a user who already holds a shorter (hence, more inclusive) permission which starts with an asterisk, and ultimately, by a user who holds a permission with domain '*'. As long as the system operators refrain from storing that string anywhere as a permission domain, no user can ever hold a completely unanchored permission.

KEYWORD RETRIEVAL USAGE

The usefulness of the permission domain PROJ3.*.DOCUMENTATION.*, regardless of where the component 'DOCUMENTATION' might appear in the names (see the earlier section on Quasi-Relational Usage), is a case of the retrieval-by-keyword aspect of this naming scheme mentioned in the Introduction. But we can adduce a couple of clearer examples:

A simple message system can be constructed from conventions for the use of this feature. For instance, let's assume that

messages are kept in the subspace MAIL.*, that in the name of each message-object the name component immediately following 'MAIL' identifies the sender of the message, and that the remaining name components identify the addressees of the message, except that the last component is a message ID number (to distinguish multiple messages from the same sender to the same set of addressees). Then user Chris would have the permissions:

```
MAIL.*.CHRIS.*      (See, Read)
MAIL.CHRIS.*        (Enter, Delete, (and perhaps Write))
```

Chris could then check for the existence of mail by commanding

```
SHOW NAMES MAIL.*.CHRIS.*
```

In actual practice, though, there would be a mail tool to handle the generation of message IDs, to mark messages as each addressee read them (perhaps by dropping its caller's identifier from the namestring), and provide all the other services of a mail program.

A more realistic example, perhaps, would be the use of the keyword retrieval abilities of this naming scheme for actual keyword retrieval:

A namespace is reserved for a literature data base, where the namestrings in the space are constructed as a set of applicable desriptors (index terms, ...), plus a document number for uniqueness. The object itself might be the entire document, or a short file giving information for obtaining the actual document.

A query of this data base to find information on microprocessors with multi-tasking abilities would then be of the form:

```
SHOW NAMES DBAS.*.MICROPROCESSOR.*.MULTI_TASKING.*
```

For such a retrieval scheme to work, two conventions would have to be observed:

- As in any keyword-index system, the indexers and the users must employ the same set of descriptors;
- Peculiarly to the present system, the set of descriptors in a namestring or in a query spec must appear in a standard order -- alphabetic, for instance -- so that each query component will be checked at a time when it might occur.

4.9 NAMED ATTRIBUTES

The objects in the system will have a number of properties in addition to their names which can usefully be kept in their catalogue entries. Now, the normal working of name-interpretation in this system requires fetching the catalogue entries for each member of the set of candidates determined by a spec, in order to check the entry's namestring against the spec and the user's permissions. Therefore, it is easy to use these other properties for performing additional checks as part of the "winnowing" process.

The properties which can be used in this checking process are called the named attributes of an object, and (in NSW) include:

Date and Time of Creation (Entry of this name into catalogue)
Creator's Identification
Date and Time of Last Change (Assignment to present object)
Last Changer's Identification
Hosts where this object is stored/executed/attached
General Type of object (Device, System Tool, Text File, ...)

For the sake of having a concrete syntax to use in examples, we shall say that these attributes have two-letter names ('TC', 'CR', 'TW', 'WR', 'HO', 'GT'), and the specification for checking an attribute is a construction of the form

<attribute_name> <relational_operator> <value_string>

A number of these specifications, separated by semicolons ';' can be added to the end of a spec, with a slash '/' separating the spec proper from the attribute specification.

Thus, for example, the command

```
SHOW NAMES SYSTEM.*.TOOL.*/HO=RADC-20;TC>831010
```

will yield a list of all the recently-added system tool programs that run on RADC-TOPS20.

The following characterization of the difference between name components and named attributes arose, as a kind of folklore design principle with no satisfactory theoretical basis, in the course of NSW's evolution:

- Attributes are "understood" by the system -- their values are calculated by some trusted component -- and the system "believes" them -- if a file's HO attribute says 'RADC-20', it is RADC-TOPS20's NSW File Package agent process which will be asked to deliver a copy of the file.

- On the other hand, name components are meaningful to the human user, but the system can only check them for order of appearance and identity. The user may choose to include the identifier 'RADC-20' as one of the name components of a file, but that doesn't mean that the file is really stored on RADC-TOPS20.

- An object's name part should uniquely identify it. That is, no two objects should have exactly the same sequence of name components and be discriminable only on the basis of the values of attributes.

It would certainly be possible to relax these strictures if that seemed desirable -- the values of selected attributes could be kept in inverted lists, which could then be used in the name-lookup process; and users could be allowed to define new attributes, whose values could be used in subspace selection.

4.10 CONTEXTS

As we've remarked, it follows from the fact that the system is to contain a very large number of objects (and from the psychological usefulness of giving similar names to objects which "belong together") that the names of the objects will tend to be long. This makes for an awkward human interface, and techniques must be found for decreasing the amount of typing a user must do to refer to objects -- especially the ones most frequently handled.

In pursuit of this aim, the system includes a feature of named contexts, another use for specs. The user can declare an identifier to name a subspace, which he defines by means of a spec. Thereafter, a spec preceded by that context name and a colon ':' will be interpreted relative to the context namespace. For example, after the user defines

```
WK: = PROJ3.SENSOR.SOFTWARE.IMAGE.*
```

the set of candidates retrieved in that context for any spec will be intersected with that subspace -- effectively, if the user requests

```
WK:*.REDUCTION.*
```

the system will respond as if he had typed

```
PROJ3.SENSOR.SOFTWARE.IMAGE.*.REDUCTION.*
```

This description of a context definition is incomplete and over-simple. A context can actually be defined by a list of specs and other contexts:

- If the list separator used is '+', a true union of the namespaces defined by the elements of the list is intended -- this is called a "union context";

- If the list separator used is ',', a sequential, "first-non-empty" search is intended -- this is called a "serial context".

For illustration: Project 3's documentation chief would reasonably have contexts

```
MANS: = PROJ3.*.MANUAL.* + PROJ3.*.MAN.DRAFT.*
DOCS: = PROJ3.*.DESCRIPTION.* + PROJ3.*.GUIDE.* + MANS:
```

and then issue a command

```
SHOW NAMES DOCS:*.TSET
```

to obtain a list of project documents that were ready for typesetting.

The serial -- comma-separated -- context is reminiscent of the TOPS-20 logical name feature, and the Unix/Multics "search path" construct. Retrieval from <ctxt>:<spec>, where <ctxt> is a serial context, works (conceptually) as follows:

1. Construct a candidate list of all catalogue entries retrieved under <spec>;
2. Intersect the candidate list with the namespace of the first/next item on the <ctxt> list;
3. If this intersection is empty, go back to step (2) using the next <ctxt> item, or return an empty result if there is no next item;
4. If this intersection is not empty, return it as the retrieval result (and look no further).

Indeed, as with search paths in other systems, this feature would be used most frequently for disambiguating the name of a program to be run; but it can be used for any type of access. A new programmer on the project might be given a context

HELP: = PROJ3.*.HELP.*, PUBLIC.*.HELP.*, SYS:*.HELP.*

so that he could ask for help on <any_subject> by typing

SHOW NAMES (of) FILES HELP:*.<any_subject>.*

and then reading the file(s), if any, whose names were returned. Notice that this returns, for the particular subject, only the help file(s) in the subspace "closest" to Project 3; if our new programmer wanted to see all the help files available to him -- on a single subject, or in general -- he would need a union context, not this serial one.

Implementation Note:

There are several interesting alternative choices one might make in implementing the context feature.

The above sketches of the use of contexts in retrieval described the process as first: collecting a candidate list from the "raw" spec; second: pruning that list according to the context definition; and finally (implicitly) further culling the list according to the user's permissions.

Clearly, these operations may be performed in any order which produces the correct result: the intersection of the spec with the union of the permissions and with the union of the context terms (or the first K context terms, if the context is serial). Hence, in cases where the context is liable to involve more subspaces than the permissions do -- as might well be true for the "dangerous" permissions (Enter, Delete, Write) -- it would be more efficient to cull the candidate list according to the permissions before passing it through the context verification.

Another possible refinement is to examine each "simple" context -- a single subspace-designator appearing as an element in a compound context -- with respect to the set of permissions; if it happens to be covered by some permission (as will usually

be the case), it can be specially marked so that the retrieval process will skip the permissions check for those items which that simple context contributes to the final result.

4.11 THE FORM OF SIMPLE CONTEXTS

An interesting design controversy arose during NSW's later development concerning the desirability of restricting the syntactic form allowed for (simple) contexts, for the sake of providing the user a simpler conceptual model.

If no restriction is put on the form of a context, then there is in general no single spec which can express the intersection of the namespace denoted by the context and the one denoted by the spec-part of a request. As the simplest example, assume the following context definition and request:

```
SPS: = PROJ3.SENSOR.*.SOURCE.*
SHOW NAMES SPS:*.IMAGE.*
```

The request asks for the set of all object names which lie in the namespace defined by SPS: and which also contain the name component 'IMAGE'. This amounts to asking for the union of the namespaces

```
PROJ3.SENSOR.*.SOURCE.*.IMAGE.*
PROJ3.SENSOR.*.IMAGE.*.SOURCE.*
```

since there is nothing in the definition and request which constrains the order of appearance of the unanchored components.

Note that this is not an implementation problem: the retrieval first returns all catalogue entries whose names contain all of the components in the context and in the spec, taken together (four name components, in the example). Then each entry is tested for being covered both by the context and by the request spec, thus effecting the intersection of those two namespaces.

The faction opposing restrictions on the form of simple contexts argued: "The user will soon learn to think of contexts (even very complex ones) as simply naming different domains of the total namespace in which he is interested. 'SPS' will mean 'Sensor Program Sources' to him, and he will formulate the request above when he wants to examine the programs in that area which might have to do with image handling."

The other faction proposed that simple contexts be restricted to contain only one asterisk. Under this rule the context definition above would be illegal; instead, knowing that in this project 'SOURCE' always appears as the last component of a program file name (at least when it is used to designate the source-language form of a program), the user would define

```
STS: = PROJ3.SENSOR.*.SOURCE
```

and then, the request

```
SHOW NAMES STS:*.IMAGE.*
```

would be equivalent to the uncontexted request

```
SHOW NAMES PROJ3.SENSOR.*.IMAGE.*.SOURCE
```

This faction argued that the user always has in mind a framework of complete object names, and chooses contexts simply as an abbreviation mechanism for filling in slots in this framework. Put another way: when the user issues a request, he has a mental image of the object names he wants retrieved, and strives to find the most efficient way to communicate that image to the system. When context definitions and request specs get complicated, he will no longer be able to envision what object names will be returned.

The implementation can take advantage of this restriction by literally embedding the request spec in the context with string operations, then using the result as the single covering pattern for checking individual namestrings.

The reader is urged, as an exercise, to work out the implications under each of the two doctrines, of these situations:

- The context definition and the request spec have a name component in common;
- The "request" spec is actually being used to give a new name to an object;
- The request spec and the context are incompatible, e.g., they specify different last components.

To complete the story, the latest version of the NSW system did adopt the restricted form for contexts, for these three reasons:

1. It was more consistent with the use of contexts in earlier versions of NSW;
2. The faction in favor of the restriction appeared to feel more strongly about the question than did the non-restrictors;
3. The implementation was faster, as described above.

4.12 DEFAULT CONTEXT, NOTATIONS

In the continuing effort to shorten the names the user has to type, we have reached the stage where he will probably not need to use more than a two-part name for any single object he refers to frequently: a context name, followed by some name component which is distinctive within that context --

```
WORK:*.PLIF.*
```

But two parts is still one too many for the most frequent uses, so the system provides for a default context which need not be named:

: = WORK

Then the command 'DELETE :*.PLIF.*' would have the same meaning as 'DELETE WORK:*.PLIF.*'.

The default context functions as a working directory plus search path; explicit contexts are used for less frequent accesses to areas outside this working namespace; and specs without a context are used for wide-ranging browsing in the system, or in circumstances where the user is given the full name of an object, and chooses to access it directly, without stopping to check which of his defined contexts would allow the shortest possible spec.

This default context notation is still somewhat awkward, and in NSW an alternative approcah was adopted: Since specs using the default context are much more frequent than specs with no context, a prefixed '$' was adopted to signify that no context was to be used, and the absence of a prefix indicated that the default context was in effect.

4.13 OWN SPACE

A simple application of the namespace-structuring doctrines discussed so far would lead to a system which made use of only a few of the billions of possible first name components -- a couple for each project, perhaps, plus some public areas (NEWS.*, BBOARD.*, MAIL.*, SYSTEM.*, HELP.*, etc.). To make better use of the namespace, and to help make object names shorter, the system allows any user (who has not been denied the right to do so) to have some number of private namespaces, each with a first component of his own choosing.

The inclusion of this feature provides an illustrative scenario of the exercise of the naming and permission schemes: When an OWN space is created -- say, one with initial name component 'KS' -- a catalogue entry with the name 'KS._OWN' is created. In normal use, the underscore character '_' is permissible within name-component identifiers, but not in initial position, so this name cannot conflict with any user-generated name. The catalogue entry for KS._OWN doesn't point to any

physical object, but records the identity of the user who created that OWN space. When another user sets out to create an OWN space for himself, he submits the name component he would like to use; the system concatenates that identifier with '_OWN' and does a normal retrieval. If no entries are found, the name is acceptable, and is assigned to the user requesting it; if one is found, the user is told that that identifier is already in use as an OWNspace name, and is asked to choose another.

When an OWN space is created, the user receives an all-rights permission to that entire namespace -- KS.* in the example above -- and the guarantee that no one else in the system has ANY rights which cover that space, initially. The user may, of course, proceed to give out sub-rights in that space, if he chooses.

There are pragmatic considerations, of course: it's probably advisable to protect (by pre-loading the catalogue) a number of popular short words and acronyms from usurpation by inappropriate users. In the NSW, since it lived on the Arpanet, a user who requested an OWN space name component of four or fewer characters was asked to affirm that that identifier was his or his organization's official NIC Ident [DIREC 83].

Since other users will not have even See rights to an OWN space, how can anyone know it exists? Well, anyone who holds a permission with See rights for the space *._OWN can examine the catalogue entries to see which OWN spaces exist, and who owns each one. It would seem to be a matter of organizational philosophy to decide whether everyone, or only system maintenance operators, should hold such a permission.

Note, in passing, that if someone holds a (See, Read) permission to *._OWN, he can see the catalogue entry named 'KS._OWN' and deduce that the namespace KS.* exists and that I own it; but this doesn't give him any rights within the space KS.* -- he can't even tell whether there are any objects in the space at all!

OWN spaces and contexts are complementary features helping to shorten typed specs. OWN spaces provide shorter names and

privacy in the user's personal area; contexts, which are easy to define and change, provide shorter names for "connecting" to more widely-shared namespaces. There is no reason for the user not to set his default context to (some subspace of) his OWN space, if that is the most convenient.

4.14 SINGULAR AND PLURAL SPECS

It was stated near the beginning of this paper that designating a single object as an argument of some operation is done by designating a set of objects which happens to have only one member. That's well and good, but there will be cases where the user will want to specify whether a command he is giving is to be executed on every element of a set or whether he intends just to abbreviate a name by using a spec which he hopes is unique within the context. Rather than encoding this difference in the name of the operation (e.g., "DELETE" vs "MULTIPLE DELETE"), we have chosen to represent it in the syntactic form of the argument.

Specs of the form we have been using are actually plural specs, as long as they contain at least one asterisk; they are used to designate the set of objects in the namespace they define, just as we have discussed. Thus,

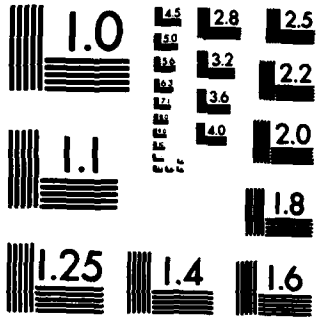
```
DELETE WS:*.REL
```

is an instruction to delete all the objects within the context WS: whose names end with the component 'REL'.

Singular specs are similar, except that the "wild card" symbol for omitted name components is an ellipsis '...' rather than an asterisk. In the mind of the user, a singular spec is an abbreviated designation of a single object, NOT a designation of a set of objects. But it does, of course, designate a set: if that set turns out to have more than one member, the spec was ambiguous and will have to be "disambiguated" by the user. Thus,

```
DELETE WS:...REL
```

AD-A146 737 NSW (NATIONAL SOFTWARE WORKS) LESSONS LEARNED(U) 2/2
MASSACHUSETTS COMPUTER ASSOCIATES INC WAKEFIELD
C MUNTZ ET AL. MAY 84 CADD-8312-3001 RADC-TR-84-90
UNCLASSIFIED F30602-83-C-0040 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

is an instruction to delete the object within the context WS: whose name ends with the component 'REL'. If there are no such objects, the user will be told; and if there are many such objects, the user will be presented with a numbered list of the objects, and asked to choose the one to be deleted. If this list is longer than some settable threshold, the user will first be told that fact and given the choice of revising his spec or seeing the list (or giving up entirely).

In summary then, if a designator string contains

- NO asterisks and NO ellipses, it is a full object name;
- NO asterisks but SOME ellipses, it is a singular spec, intended to abbreviate a particular namestring;
- SOME asterisks and NO ellipses, it is a plural spec, intended to designate a sub-namespace and the set of objects whose names are in that space;
- SOME asterisks and SOME ellipses, it is a plural spec because of the presence of an asterisk -- but what meaning can the presence of an ellipsis (instead of, say, another asterisk) carry?

The answer has mostly to do with the use of plural specs in operations, which we shall take up in a moment. The idea is that an asterisk signifies that we "do care" about the name-component strings that appear in its place in object names, and the ellipsis signifies that we "don't care". The only circumstance so far discussed in which this distinction could profitably be used is in the restricted context debate: Contexts may be restricted to contain exactly one asterisk (marking the place to insert the request spec) while ellipses would mark other regions of the namespace specifier which don't contribute to the set specification.

Returning to the example of the restricted spec, our user "knew" that in his project the name component 'SOURCE' was always placed at the end of an object name, and defined his context as 'PROJ3.SENSOR.*.SOURCE'. More realistically, he would realize that the programmers are liable to "temporarily" tack on another

component like 'OLD', 'FRAG', 'COPY'. So to get a good working context for examining the source modules in the various divisions of the sensor subproject, he could define it as 'PROJ3.SENSOR.*.SOURCE...', and pick up the deviant names as well.

4.15 PLURAL OPERATIONS

The reader will perhaps have noticed that the only examples we have given of commands with plural operands have been single-operand ones -- namely, SHOW NAMES and DELETE. For such commands, there's no question of how to interpret the plural spec:

```
SHOW NAMES PROJ3.*.DOCUMENTATION.*.TSET
```

will show the full namestrings of all existing objects in the system whose names start with the component 'PROJ3', end with the component 'TSET', and have the component 'DOCUMENTATION' somewhere in between. The order in which the names returned from this request would be displayed is determined by convention: alphabetically, for instance, or in the order in which the name interpretation function actually finds the names.

For operations with more than one operand, the interpretation becomes more delicate (just as 't does in other naming schemes). Consider a programmer who has finished writing and preliminary testing of a package of programs to be used with the Actuator assembly of Project 3. He has been working in a subspace of his OWN space, and has his default context appropriately defined:

```
: = CAM3.ACTUATOR.*
```

Now he is ready to "release" his programs for testing, by COPYing them into the Testbed space of PROJ3.ACTUATOR.*

```
ACTEST: = PROJ3.ACTUATOR.TESTBED.*
```

He could also have chosen to RENAME the files, thereby conceptually moving them into the Testbed space; the choice would depend on project doctrines, or his style of debugging. We'll also assume that project doctrines require programs for testing to be recompiled in the Testbed environment, so that he needs only to copy over the source-language forms of the programs. He should be able to utter the command

```
COPY (from) *.SOURCE (to) ACTEST:*.SOURCE
```

The intended operation is clear. The name-interpretation function will return the set of all existing names of the form

```
CAM3.ACTUATOR.*.SOURCE
```

whereupon, for each of these names, the sequence of name components mapping into the '*' will be copied into the place of the '*' in a new name of the form

```
PROJ3.ACTUATOR.TESTBED.*.SOURCE
```

Thus a copy of the program file CAM3.ACTUATOR.RHO_POS.INTEGRATOR.SOURCE will be made, and given the name 'PROJ3.ACTUATOR.TESTBED.RHO_POS.INTEGRATOR.SOURCE'; and so on for all the other objects in the source namespace.

The interpretation of a source/destination pair of plural operands, where there is only one asterisk in each, is clear enough. If there are several asterisks, the convention of identifying asterisks by their relative ordering is a plausible extension of the simpler case:

In our example, assume that the compilation tools of the project required that the last name component of source files be an indicator of the language in which the source is written -- .ADA, .FTN, .ASEMB. If our programmer had programs in all three languages, he would have commanded:

COPY (from) :*.SOURCE.* (to) ACTEST:*.SOURCE.*

A more general and complex scheme was designed and discussed in the course of the NSW project, but did not get close enough to actual implementation to be considered under the rules for the content of this paper. As an advanced exercise for the interested reader, it can be stated in terms of compact generalities:

> Adopt a scheme for identifying individual asterisks, e.g.: 'COPY (from) AB.*2.CD.*1.EF (to) RS.*1.*2.TUV'. Construct the destination names by substituting for the tagged asterisks the name-component sequences corresponding to the same tagged asterisks in the source spec. This yields a very flexible name-rewriting facility. For single-operand commands, tagged asterisks can be used to dictate the order of presentation of the names covered by the spec, e.g.: 'SHOW DESCRIPTORS PROJ3.*2.DOCUMENTATION.*3/WR:*1' would sort the output display first by the identification of the person doing the most recent modification, and alphabetically within groups having the same WR value.

We might venture the remark in passing that most file systems don't have a very flexible multiple name-changing scheme applying to sets of file names, so there aren't very good standards for judging whether a particular proposal is overly complex. The extension described above does indeed strike people as complicated.

4.16 THE FINAL STATE OF NSW

In actual use in NSW, the object naming scheme evolved "upward" from the idea of abbreviating single file names by using just one or two name components of high discriminative value. The implementation was built upon a pre-designed general information-retrieval system which kept inverted indices for each individual identifier used as a name component, and so the implementation technique of skipping collision resolution, mentioned early in this paper, was not used.

The evolution toward plural specs as operands of users' commands was incomplete; the feature was supported only for the commands SHOW NAMES and SHOW DESCRIPTORS. The DELETE command with a plural argument was waiting for a final decision on the user interaction patterns (since NSW did not have an "undelete" operation). Plural specs in source/target commands were in the design stage only.

As mentioned, contexts were restricted to containing a single asterisk, and the "mixed" form -- allowing ellipses along with the asterisk -- was not supported. Named contexts were not offered to the user, although the record-keeping and interpretation mechanisms were present. The unnamed default context feature was provided, with a separate operation for setting or changing it, or restoring it from the user's profile.

The only <relational operator> allowed in specifying the values of named attributes was equality, writtem ':'. A form of less-than operator was provided by accepting initial-substring matches on the specified value of date-time attributes.

4.17 ACKNOWLEDGEMENTS

In the course of the many years' development of NSW, well over half of the technical personnel of Massachusetts Computer Associates ("Compass") played some part in the project, and most of them were involved at one time or another with the "file system". The original design, started in 1974, was principally the work of Stephen Warshall and Robert Millstein, with some participation by the present author. The first implementation was done by Stuart Schaffner and Regina Bolduc. The major developers during the intermediate stages were Ross Faneuf and Suzanne Sluizer. A major reconsideration of the design occurred later, with Warshall and Sattley of Compass working with Robert Thomas and Richard Schantz of Bolt Beranek and Newman ("BBN"). Still further discussions of the design were carried on with Schantz and William McGregor of BBN and Charles Muntz, Mark Marcus, and Sattley of Compass. None of the other individuals mentioned above deserves to be taxed with supporting any point of view expressed in this paper.

REFERENCES

[Abraham 80] Abraham, Steven M. and Dalal, Yogen K.
Techniques for Decentralized Management of Distributed Systems.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[AIEK 82] Intermetrics.
Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database, Type B5.
Technical Report IR-678-1, Intermetrics Inc., June, 1982.

[Apollo 81a] Apollo Domain Architecture
Apollo Computer Inc., Chelmsford, MA, 1981.

[Apollo 81b] Apollo System User's Guide
Apollo Computer Inc., Chelmsford, MA, 1981.

[Clark 80] Clark, David D. and Svobodova, Liba.
Design of Distributed Systems Supporting Local Autonomy.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[Cohen 82] Cohen, Danny.
Internet Mail Forwarding.
In COMPCON Spring 82 Digest of Papers. IEEE Computer Society, February, 1982.

[Dalal 82] Dalal, Yogen K.
Use of Multiple Networks in Xerox's Network.
In COMPCON Spring 82 Digest of Papers. IEEE Computer Society, February, 1982.

[DIREC 83] NIC.
Arpanet Directory.
Technical Report NIC 49000, Network Information Center, SRI International, 1983.

[Donnelley 79] Donnelley, James E.
components of a Network Operating System.
Computer Networks 3(6):389-399, December, 1979.

[Fletcher 80] Fletcher, John G. and Watson, Richard W.
Service Support in a Network Operating System.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[Fletcher 82] Fletcher, John G. and Watson, Richard W.
An Overview of LINCS Architecture.
Technical Report, Lawrence Livermore Laboratory, November, 1982.

[Gehringer 82] Gehringer, Edward F., Jones, Anita K. and Segall, Zary Z.
The Cm* Testbed.
Computer 15(10):40-53, October, 1982.

[Hoffman 82] Hoffman, Morton D., MacGregor, William I., Schantz, Richard E. and Thomas, Robert H.
DOS Functional Definition and System Concepts Draft.
Technical Report, Bolt Beranek and Newman Inc., April, 1982.

[Landweber 82] Landweber, Lawrence H. and Solomon, Marvin H.
Use of Multiple Networks in CSNET.
In COMPCON Spring 82 Digest of Papers. IEEE Computer Society, February, 1982.

[Lantz 80] Lantz, Keith A.
Uniform Interfaces for Distributed Systems.
PhD thesis, University of Rochester, May, 1980.

[Lantz 82] Lantz, Keith A., Gradischnig, Klaus D., Feldman, Jerome A., and Rashid, Richard F.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[Lauer 78] Lauer, Hugh C. and Needham, Roger M.
On the Duality of Operating Systems Structures.
In Proceedings second International Symposium on Operating Systems. IRIA, April, 1978.
Reprinted in Operating Systems Review, 13, 2, April 1979, pp. 3-19.

[Lazowska 81] Lazowska, Edward D., Levy, Henry M., Almes, Guy T., Fischer, Michael J., Fowler, Robert J., Vestal and Stephen C.
The Architecture of the Eden System.
In Proceedings of the Eighth Symposium on Operating Systems Principles. ACM/SIGOPS, December, 1981.

[Liskov 82] Liskov, Barbara and Scheifler, Robert.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
In Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages. ACM/SIGACT, ACM/SIGPLAN, January, 1982.

[Lister 79] Lister, Andrew M.
Fundamentals of Operating Systems.
Springer-Verlag New York Inc., New York, NY, 1979, pages 9.

[MacGregor 82] MacGregor, William I. and Tappan, Daniel C.
The Cronus Virtual Local Network.
Technical Report, Bolt Beranek and Newman Inc., August, 1982.

[Mamrak 82a] Mamrak, S. A., Gomez, J., Janardan, S. and Nicholas, C.
Guest Layering Distributed Processing Support on Local Operating Systems.
In Proceedings 3rd International Conference on Distributed Computer Systems. IEEE Computer Society, October, 1982.

[Mamrak 82b] Mamrak, S., Dunnington, R. and Shaffer, B.
Installing Existing Tools in a Distributed Processing System.
submitted to ACM Transactions on Computer Systems , 1982.
Copy dated March 1983 -- could be an updated working paper?

[Mamrak 82c] Mamrak, S. A., Kuo, J. and Soni, D.
Supporting Existing Tools in Distributed Processig Systems: the Conversion Problem.
In Proceedings 3rd International Conference on Distributed Computer Systems. IEEE Computer Society, October, 1982.

[Mamrak 83] Mamrak, Sandra A. and Leinbaugh, Dennis.
A Progress Report on the Desperanto Research Project - Software Support for Distributed Processing.
Operating Systems Review 17(1):17-29, January, 1983.

[McGann 83] McGann, L. Brian.
Courier Protocol Accesses Services with Remote Procedure Calls.
Systems & Software , March, 1983.

[Millstein 77] Millstein, R. E.
The National Software Works: A Distributed Processing System.
In Proceedings ACM Annual Conference. ACM, 1977.

[Nelson 80] Nelson, David L.
Application Engineering on a Distributed Computer Architecture.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[Nelson 83] Nelson, David L.
Virtual Memory/Addressing Manages Token-passing Ring's Resources.
Systems & Software , March, 1983.

[NSW 83] Ata, John, Paul M. Cashman, Henrik O. Lind, Neil Ludlam, Steven A. Swernofsky, Stephen G. Toner, and Kirk Sattley.
NSW User Reference Manual.
Technical Report, Honeywell, UCLA, BBN, and Massachusetts Computer Associates, 1983.

[Peebles 80] Peebles, Richard and Dopirak, Thomas.
ADAPT: A Guest System.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[Popek 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G.
LOCUS: A Network Transparent, High Reliability Distributed System.
In Proceedings of the Eighth Symposium on Operating Systems Principles. ACM/SIGOPS, December, 1981.

[Rashid 81] Rashid, Richard F. and Robertson, George G.
Accent: A Communication Oriented Network Operating System Kernel.
In Proceedings of the Eighth Symposium on Operating Systems Principles. ACM/SIGOPS, December, 1981.

[Ritchie 78] Ritchie, D. M. and Thompson, K.
The UNIX Timesharing System.
The Bell System Technical Journal 57(6, part 2):1905-1930, July-August, 1978.

[Rowe 82] Rowe, Lawrence A. and Birman, Kenneth P.
A Local Network Based on the UNIX Operating System.
IEEE Transactions on Software Engineering SE-8(2):137-146, March, 1982.

[Schantz 82] Schantz, R., Burke, E., Geyer, S., Hoffman, M., Lake, A., Pogran, K., Tappan, D., Thomas, R., Toner, S. and MacGregor, W.
Cronus, A Distributed Operating System: Interim Technical Report No. 1.
Technical Report, Bolt Beranek and Newman Inc., July, 1982.

[Schantz 83] Schantz, R., Woznik, B., Bono, G., Burke, E., Geyer, S., Hoffman, M., MacGregor, W., Sands, R., Thomas, R. and Toner, S.
Cronus, A Distributed Operating System: Interim Technical Report No. 2.
Technical Report, Bolt Beranek and Newman Inc., February, 1983.

[Schuman 81] Sattley, Joanne Z., Kirk Sattley, Stuart C. Schaffner, Stephen C. Schuman, Edmund M. Clarke Jr.
Ada Distributed Application Prototyping Technique.
Technical Report CA-8110-0101, Massachusetts Computer Associates and Harvard University, 1981.

[Smith 81] Smith, Reid G. and Davis, Randall.
Frameworks for Cooperation in Distributed Problem Solving.
IEEE Transactions on Systems, Man and Cybernetics SMC-11(1):61-70, January, 1981.

[Tanenbaum 81] Tanenbaum, Andrew S.
Computer Networks.
Prentice Hall, Englewood Cliffs, NJ, 1981, pages 476-483.

[Thomas 73] Thomas, Robert H.
A Resource Sharing Executive for the ARPANET.
In Proceedings National Computer Conference. IFIPS, June, 1973.

[Ullman 82] Ullman, Jeffrey D.
Principles of Database Systems.
Computer Science Press, 1982.

[Ward 80] Ward, Stephen A.
TRIX: A Network-oriented Operating System.
In COMPCON Spring 80 Digest of Papers. IEEE Computer Society, February, 1980.

[Warshall 80] Warshall, Stephen.
A Theory of Accountability.
Technical Report, Massachusetts Computer Associates, November, 1980.

[Watson 79] Watson, Richard W. and Fletcher, John G.
An Architecture for Support of Network Operating System Services.
Technical Report, Lawrence Livermore Laboratory, August, 1979.

[Watson 82] Watson, Richard W.
LINCS Session, Presentation, Common Application/Service Protocols Common Across Applications/Service.
Technical Report, Lawrence Livermore Laboratory, March, 1982.
Working Draft.

[Watson 83] Watson, Richard W.
Distributed System Architecture Model.
In Lampson, B. W., Paul, M. and Siegert, H. J. (editor), Distributed Systems Architecture and Implementation: An Advanced Course, chapter 2. Springer-Verlag, Berlin, 1983.

[White 76] White, James E.
A High-level Framework for Network-based Resource Sharing.
In AFIPS Conference Proceedings, National Computer Conference. AFIPS, June, 1976.

[Xerox 81] Courier: The Remote Procedure Call Protocol
Xerox Corporation, Stamford, Connecticut, 1981.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

11-84

DTIC